DISSERTATION

"Bit-level non-destructive arbitration of CAN controllers"

By:

Kwong Lai Yeen (1473)

Dissertation submitted in partial fulfilment of
the requirements for the
Bachelor of Engineering (Hons)
(Electrical and Electronics Engineering)

JUNE 2004

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

Bit-level non-destructive arbitration of CAN controllers

by

Kwong Lai Yeen

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Approved by,

(Mr. Abu Bakar Sayuti)
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

June 2004

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

KWONG LAI YEEN

ABSTRACT

This report is written as part of the requirement of Final Year Project in progress. The title; "Bit-level non-destructive arbitration of CAN controllers" was selected by the author from a selection of titles provided by lecturers and approved by the Final Year Project (FYP) committee.

Chapter 1 of the report presented a brief overview on the project scope and concepts applied. It gave some introduction and a brief history on Controller Area Network (CAN). The problem statement which leads to the implementation of the project has also been highlighted. The objective of the project has also been defined in this section in which the main aim of this project is have an FPGA implementation of a CAN controller which will be able to demonstrate the non-destructive arbitration operation when sending messages across the bus. Chapter 2 of the report discussed more on CAN in general. It explained on the CAN protocol and the principle used in the network. CAN in general is divided into three layers which is the Object Layer, Physical Layer and Transfer Layer. Each layer has its corresponding tasks or functionality in data/message handling within the network. In network data transmission, CAN uses a method known as Carrier Sense, Multiple Access with Collision Detect (CSMA/CD) but with the enhanced capability of non-destructive bitwise arbitration to handle message collision to deliver maximum use of the available capacity of the bus.

In Chapter 3, the methodology used in implementing the project has been identified. The methodology schedule is based on the Gantt chart (Appendix A). The FPGA design flow used to program into the design into the FPGA chip has also been presented. In Chapter 4, some discussions and findings of CAN especially in the bit-level arbitration process of CAN has been discussed. The Register Transfer Level (RTL) simulation results and the Logic Analyzer captured output waveform has been analyzed and verified. The last section consists of the conclusion and some recommendations to improve on the design.

ACKNOWLEDGEMENT

This project would not have been possible without the help of a number of people, and the author would like to express her utmost gratitude to all of them.

The author would like to express her foremost gratitude to her supervisor, Mr. Abu Bakar Sayuti for his guidance and endless supports in the course of this project. Being under his supervision has been an irreplaceable experience; Mr. Abu Bakar has continuously monitored her progress and guided her throughout the duration of the project. His comments, critiques and suggestions were given serious consideration and were invaluable in determining the final outcome of the project.

Heartfelt gratitude also goes to Mr. David Kong, Mr. Ho Tatt Wei and Mr. Ng Kiat Hong, the author's fellow course mates who have been very helpful in providing basic tutelage in high-level programming to the author. Thank you very much for their support.

The author would also like to extend her sincerest thanks to Mr. Goh Teik Ming, the authors' good friend for providing valuable insights, ideas and assistance in one way or another throughout the duration of the project. His many useful comments and support has indeed helped the author in completing her project successfully. Also, the author would like to express her gratitude to the UTP Electrical and Electronics Lab technician especially Encik Musa bin Mohd Yusof for his valuable tips and assistance from time to time.

Last but not least, the author would also like to thanks her family for their continuous love and support, for which is a source of strength and motivation to the author. Finally, a very big thank you to everyone who has directly or indirectly assisted the author in different aspects throughout the development of her project. Without their constant guidance, supervision and encouragement, this project would not have been successfully completed. Thank you again for making this project a thorough learning experience for the author. Your kindness will be deeply appreciated.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

This section provides some insights on the topic of interest, Controller Area Network (CAN). In addition, the problem statement of the project has also being defined. Besides, the objectives of the project and the scope of study have also being provided in this section.

1.1 BACKGROUND OF STUDY

1.1.1 Brief History of CAN

Controller Area Network (CAN) which was developed in the year 1986 was the brainchild of Robert Bosch, a German automotive system supplier. It was initially developed for automotive industry applications to ensure a more robust serial communications for networking in vehicles. CAN is a technology designed for automobiles to be more reliable, safe and efficient while decreasing wiring harness weight and complexity within the interior of vehicle electronics. With the use of CAN, point-to-point wiring in vehicle wiring systems is gradually being replaced by one serial bus connecting all control systems. Besides in-vehicle applications, CAN is also being employed in the industry. It is usually used as a communication bus for message transaction in small-scaled distributed environment.

1.1.2 Introduction

Layered approach is commonly used for network applications in system implementation. This systematic approach provides standards which enables interoperability between products from different manufacturers. Similarly for CAN, a layered approached has been applied in its protocol. CAN is internationally standardized by the International Standardization Organization (ISO) and the Society of Automotive Engineers (SAE) which provide a template for this layered approach. It is called the Open Systems Interconnection (OSI) Network Layering Reference Model (As illustrated in Figure 1.1). The CAN protocol itself implements most of the lower two layers of this reference model, the Data Link Layer and the Physical Layer [4].

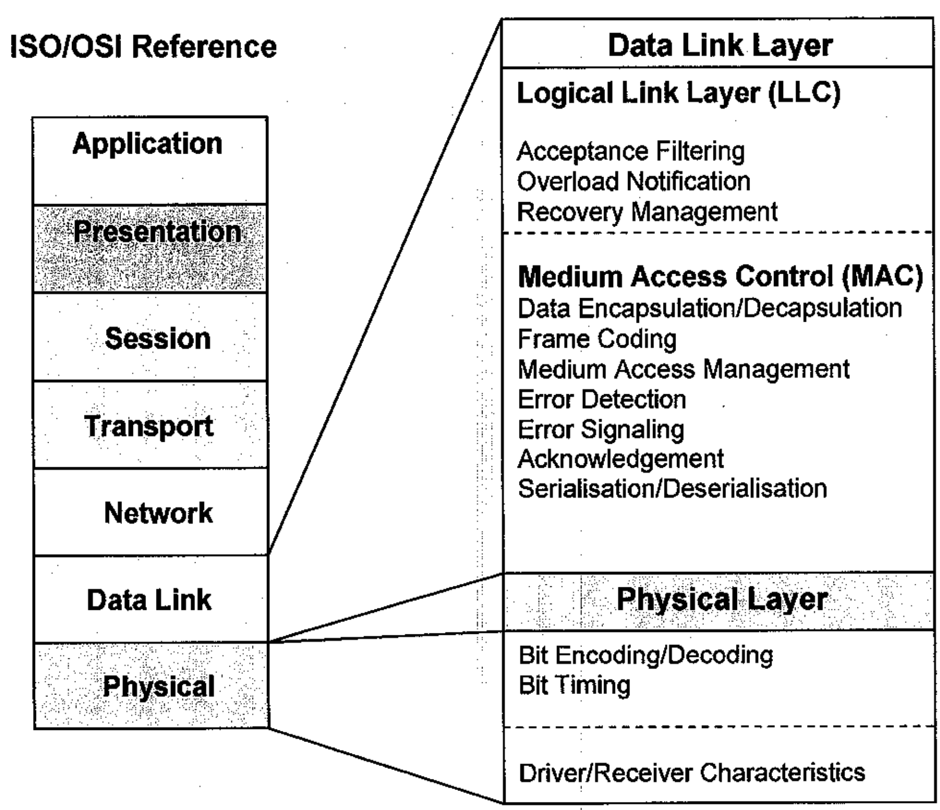


Figure 1.1: ISO/OSI Reference Model [4]

As shown in Figure 1.1, the Data Link layer of CAN is further subdivided into two sub layers, which is the Logical Link Control (LLC) and Medium Access Control (MAC) sub layers. The Data Link layer is the only layer that recognizes and understands the format of messages. This layer constructs the messages to be sent to the Physical Layer, and decodes messages received from the Physical Layer [2].

The Physical layer on the other hand, specifies the physical and electrical characteristics of the bus. It is responsible for the transfer of bits between the different nodes in a given network. It defines how signals are transmitted and therefore deals with issues like timing, encoding and synchronization of the bit stream to be transferred. This layer is usually the hardware that converts the characters of a message into electrical signals for transmitted messages. It also converts messages from electrical signals into characters for received messages. Although the other layers may be implemented in either hardware (as chip level functions) or software, the Physical layer is always "real" hardware (usually a twisted pair of wire/cable or any other medium of transmission).

1.2 PROBLEM STATEMENT

Currently, low-cost CAN controllers and interface devices are available as off-the-shelf parts manufactured by several of the leading semiconductor manufacturers such as Fujitsu, Hitachi, Intel, Texas Instruments and Phillips Semiconductors. Custom built devices and popular microcontrollers with embedded CAN controllers are also available. However, most of these CAN controllers are proprietary, and as such customization and further design evolution of the chips will require permission and consultation from respective manufacturers which in turn will incur more cost towards system development.

Besides, CAN technology is relatively new in Malaysia unlike in the United Kingdom where CAN has already received widespread used in different areas of expertise especially in automotive and industrial applications. It is hope that this project will serve as an introduction and familiarization with CAN technology in

Malaysia. The results and research work of this project will serve as a foundation for future development of CAN in the country.

1.3 OBJECTIVE AND SCOPE OF STUDY

The main objective of this project is to be able to implement a section of CAN network bus with a reasonable degree of performance. The implementation will focus on the Transfer layer of CAN (explained further in Section 2.2.) which is responsible for the bit-level non-destructive arbitration of CAN controller.

The design is an FPGA-based implementation which includes the programming of a CAN controller system onto the FPGA demo board with hardware description language like VHDL (VHSIC Hardware Description Language) as the core programming language. One of the main CAN controllers must be able to handle collisions of signals by bit-level non-destructive arbitration process which is important in eliminating message re-transmission and unnecessary network overloading. Another CAN controller in the design will compete in the usage of the network bus with the main CAN controller. This is done to ensure that the arbitration process of the CAN system can be observed and analyzed whenever one or more nodes (represented by the CAN controllers) are sending message to the bus. The output signals of the CAN controller will then be analyzed and captured with a Logic Analyzer to investigate the arbitration of signals behavior in the controller.

In order to ensure that this project will be feasible within the scope and time frame, the concentration of this project will be largely based on the implementation of the message handling and collision section of CAN. The other principal functionality of CAN like error handling and remote data transfer will not be included. This project will be implemented within two semesters where the first semester covers on the understanding of the CAN concept and VHDL modules programming. For the second semester, design flow in accordance to the Xilinx FPGA implementation has been adopted.

CHAPTER 2

LITERATURE REVIEW

This section provide more information on the CAN protocol which includes the basic principle of CAN, the three layers significant in CAN, its message format and more on the non-destructive bit-level arbitration process. The information presented is mostly obtained from relevant books and online resources. More information on each section can be obtained from the direct source in which it has been referenced to (The number enclosed within the square brackets corresponds to the referenced item in the References Section). Besides, some information on the hardware description language used for this project, VHDL is included in this section as well.

2.1 BASIC CAN PRINCIPLE

With reference to [3] and [4], CAN principle has been described in this section. CAN is an advanced serial bus system that efficiently supports distributed control systems. It is a broadcast bus that has an open, linear structure with one logic bus line and equal nodes. CAN is also a message-based protocol, not an address based protocol. As such, the messages are not transmitted from one node to another node based on addresses but the message is broadcasted to all nodes and each message is referred to by an identifier within the message itself which indicates the message content and the priority of the message. This identifier is unique throughout the network. All other nodes on the network receive the message and each performs an acceptance test on the identifier to determine if the message, and thus its content, is relevant to that particular node. If the message is relevant, it will be processed, otherwise it is ignored. Since the nodes do not have addresses, the number of nodes may be changed dynamically without disturbing the communication of the other nodes.

2.2 CAN LAYERS

In order to achieve design transparency and implementation flexibility, CAN has been subdivided into different layers. They are:-

- The Object layer
- The Transfer layer
- The Physical layer

The object layer and the transfer layer comprise all services and functions of the data link layer defined by the ISO/OSI model (As being mentioned in Section 1.1) [11].

2.2.1 Object Layer

The scope of the object layer includes:

- Finding which messages are to be transmitted.
- Deciding which messages received by the transfer layer is actually to be used.
- Providing an interface to the application layer related hardware.

2.2.2 Transfer Layer

The scope of the transfer layer mainly is the transfer protocol which includes:-

- Controlling the framing
- Performing arbitration
- Error checking and error signaling
- Fault confinement.

2.2.3 Physical Layer

The scope of the physical layer is the actual transfer of the bits between the different nodes with respect to all electrical properties. Within one network the physical layer, of course, has to be the same for all nodes. There may be, however, much freedom in selecting a physical layer.

2.3 CAN MESSAGE FRAME

With reference to [11], it is found that CAN protocol define four different types of messages (or Frames). They include:-

- Data Frame
- Remote Frame
- Error Frame
- Overload frame

The most common type of frame is a Data Frame. This is used when a node transmits information to any or all other nodes in the system. The second frame is called a Remote Frame, which is basically a Data Frame with the Remote Transmit Request (RTR) bit set. The other two frame types are for handling errors. One is called an Error Frame and the other one is called an Overload Frame. Error Frames are generated by nodes that detect any one of the many protocol errors defined by CAN. Overload errors are generated by nodes that require more time to process messages already received.

Data Frames and Remote Frames will be further explained. Data Frames consist of fields that provide additional information about the message as defined by the CAN specification. Embedded in the Data Frames are Arbitration Fields, Control Fields, Data Fields, CRC Fields, a 2-bit Acknowledge Field and an End of Frame.

The Arbitration Field is used to prioritize messages on the bus. Since the CAN protocol defines a logical 0 as the dominant state, the lower the number in the arbitration field, the higher priority the message has on the bus. The arbitration field consists of 12-bits (11 identifier bits and one RTR bit) or 32-bits (29 identifier bits, 1-bit to define the message as an extended data frame, an SRR bit which is unused, and an RTR bit), depending on whether Standard Frames or Extended Frames are being utilized. The current version of the CAN specification is Version 2.0B, which defines 29-bit identifiers. They are known as the Extended Frames. Previous versions of the CAN specification defined 11-bit identifiers which are called Standard Frames. The CAN protocol version will be explained further in Section 2.4.

The Remote Transmit Request (RTR) is used by a node when it requires information to be sent to it from another node. To accomplish an RTR, a Remote Frame is sent with the identifier of the required Data Frame. The RTR bit in the Arbitration Field is utilized to differentiate between a Remote Frame and a Data Frame. If the RTR bit is recessive, then the message is a Remote Frame. If the RTR bit is dominant, the message is a Data Frame.

The Control Field consists of six bits. The most significant bit (MSB) is the IDE bit (signifies Extended Frame) which should be dominant for Standard Data Frames. This bit determines if the message is a Standard or Extended Frame. In Extended Frames, this bit is RB1 and it is reserved. The next bit is RB0 and it is also reserved. The four least significant bits (LSB) are the Data Length Code (DLC) bits. The Data Length Code bits determine how many data bytes are included in the message. It should be noted that a Remote Frame has no data field, regardless of the value of the DLC bits.

The Data Field consists of the number of data bytes described in the Data Length Code of the Control Field. The CRC Field consists of a 15-bit CRC field and a CRC delimiter, and is used by receiving nodes to determine if transmission errors have occurred. The Acknowledge Field is utilized to indicate if the message was received correctly. Any node that has correctly received the message, regardless of whether the node processes or discards the data, puts a dominant bit on the bus in the ACK Slot bit

The last two message types are Error Frames and Overload Frames. When a node detects one of the many types of errors defined by the CAN protocol, an Error Frame occurs. Overload Frames tell the network that the node sending the Overload Frame is not ready to receive additional messages at this time, or that intermission has been violated. Figure 2.1 and Figure 2.2 shows the Data Frame and Remote Frame for a Standard CAN (Version 2.0A).

Data Frame of CAN 2.0A (Standard)

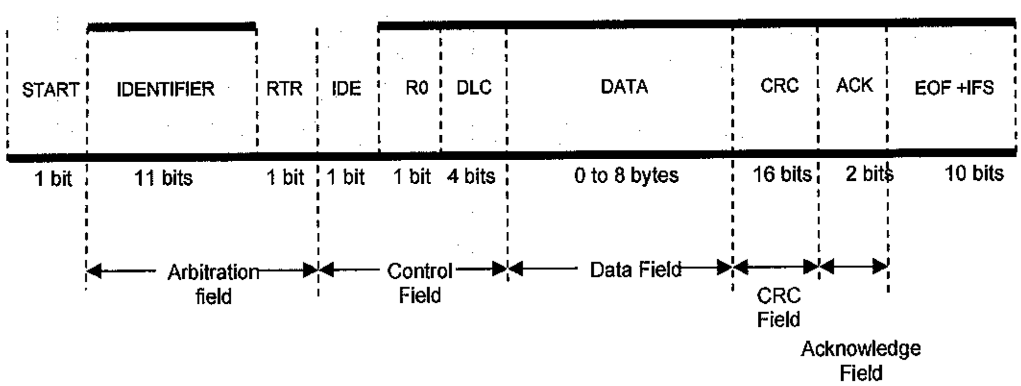


Figure 2.1: CAN Data Frame [13]

Remote Frame of CAN 2.0A (Standard)

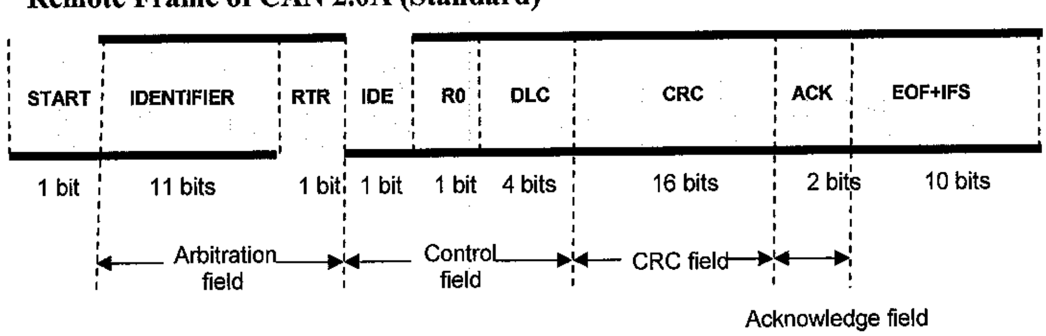


Figure 2.2: CAN Remote Frame [13]

2.4 CAN PROTOCOL VERSION

The CAN protocol supports two message frame formats, the only essential difference being in the length of the identifier. The CAN standard frame supports a length of 11 bits for the identifier, and the CAN extended frame, supports a length of 29 bits for the identifier.

2.5 DATA TRANSMISSION IN CAN

In any systems, some parameters will change more rapidly than others. It is likely that the more rapidly changing parameters need to be transmitted more frequently and, therefore, must be given a higher priority. To determine the priority of messages, CAN uses an established method known as CSMA/CD that is similar to that used in ETHERNET. However, besides the CSMA/CD technology, CAN have an enhanced capability of non-destructive bitwise arbitration to provide collision resolution, and to deliver maximum use of the available capacity of the bus.

The 'CSMA' stands for Carrier Sense Multiple Access. What this means is that every node on the network must monitor the bus for a period of no activity before trying to send a message on the bus (Carrier Sense). Also, once this period of no activity occurs, every node on the bus has an equal opportunity to transmit a message (Multiple Access). The abbreviation, 'CD' stands for Collision Detection. If two nodes on the network start transmitting at the same time, the nodes will detect the collision and take the appropriate action [2].

2.5.1 Non-Destructive Bitwise Arbitration

From [5], the following information has been further obtained. Bus access conflicts are resolved by non-destructive bit-wise arbitration in CAN in the transfer layer of the layered structure of CAN which is explained in Section 2.2. The protocol happens in accordance with the "wired-and" mechanism, by which the dominant state overwrites the recessive state. The priority of a CAN message is determined by the numerical value of its identifier. The numerical value of each message identifier (and thus the priority of the message) is assigned during the initial phase of system design. A fundamental CAN characteristic in this sense is that the lower the message number, the higher its priority. Therefore, an identifier consisting entirely of zeros is deemed to be the highest priority message.

CAN utilize binary signaling with a high and low signal state and an idle signal state that is defined as high. To transmit a logical '0' bit, a node sinks the bus state to low for one bit time. This is called a dominant bit. To transmit a logical '1' bit, the state of the line is left high for one bit time. This is called a recessive bit. Collision-avoidance begins when two or more nodes simultaneously begin to transmit the first bit of their frame-identifier.

At any time during priority arbitration, a node transmitting a dominant bit (logical 0) has a higher priority than any node transmitting a recessive bit (logical 1). A node transmitting a recessive bit effectively monitors the bus state for one bit time. Upon detection of a dominant bit transmission, this node recognizes a higher priority frame and drops out of contention. This process is repeated over the length of the identifier. Given that the frame identifiers are unique, only one node can be left in contention at the end of the bit-wise arbitration. This effectively realizes a priority arbitration mechanism wherein the identifier with the lowest numeric value has the highest priority. Figure 2.3 shows an example of arbitration process in CAN.

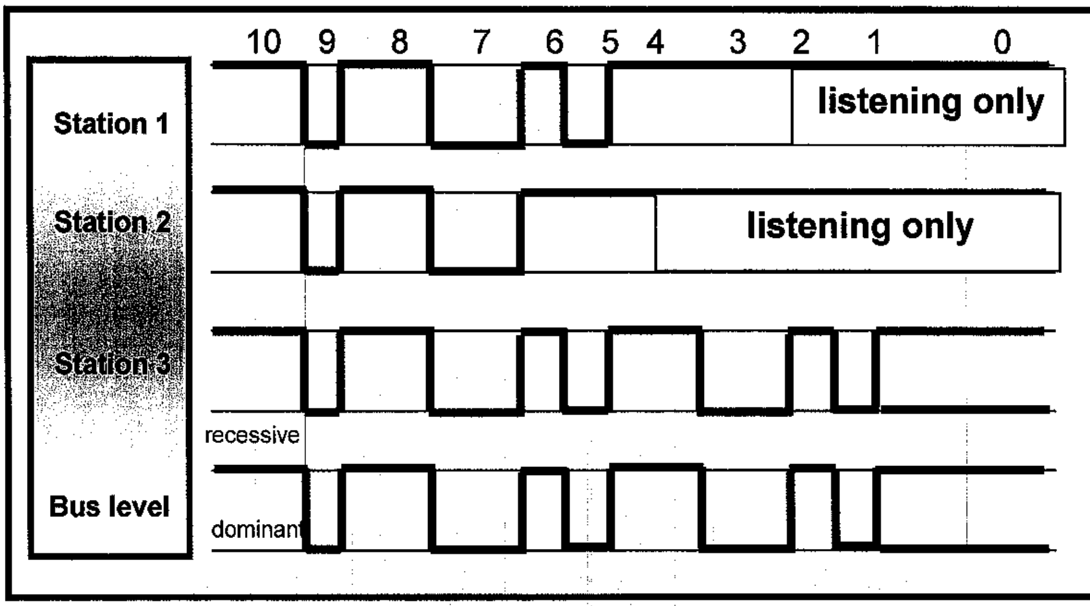


Figure 2.3: An example of CAN arbitration process [5]

From Figure 2.3, station one and station two has lost in the arbitration of signals. Station 3 which have the highest priority (lowest identifier value) is thus in the transmitter mode and is successful in transmitting the complete data frame. Station 1 and Station 2 on the other hand, has switched to receiver mode upon detection of its arbitration state. In the receiver mode, the station only "listens" to the messages and will decide whether to accept or reject the messages. Station 1 and Station 2 and will resend the message (data frame) once the bus is free again (in recessive mode).

2.5.2 The Benefits of Non-Destructive Bitwise Arbitration

Non-destructive bitwise arbitration provides bus allocation on the basis of need, and delivers efficiency benefits that cannot be gained from either fixed time schedule allocation (e.g. Token ring) or destructive bus allocation (e.g. Ethernet.). With only the maximum capacity of the bus as a speed limiting factor, CAN is indeed more superior in term of message handling across transmission medium. Outstanding transmission requests are dealt with in their order of priority, with minimum delay, and with maximum possible utilization of the available capacity of the bus [2].

2.6 VHSIC HARDWARE DESCRIPTION LANGUAGE (VHDL)

From [6], [7] and [9], the following information has been obtained. VHDL is a hardware description language that can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits. It can describe a digital system at several different levels, which is behavioral, dataflow and structural. VHDL leads naturally to a top-down design methodology in which system is first specified at a high level and tested using a simulator. The simulator is used to verify the behavior of the digital circuit prior to expensive fabrication After the system is debugged at this level, the design can be refined, eventually leading to a structural description closely related to the actual hardware description.

VHDL program is unlike any conventional program written in either Pascal or FORTRAN. In VHDL, the focus is in describing the behavior of some physical system rather than how a function is computed. The VHDL description can be used to support two complimentary processes found in the design of digital system which is simulation and synthesis. Simulation and synthesis are complementary design processes. In both cases, the specification of the behavior of the digital system is the first step to construct a VHDL model for the desired system. A VHDL simulator executes this model to mimic the behavior of the physical circuit where the behavior is described in terms of the occurrence of events and waveforms of signals. In contrast, digital circuit synthesis is the reverse process. A VHDL program is the input to a synthesis compiler that can process this description to generate the physical design of a circuit. Essentially, the synthesis compiler mimics the activities of what used to be a human chip designer job to generate a hardware design from an initial specification.

CHAPTER 3

METHODOLOGY / PROJECT WORK

This section describes the procedures and project flow used in implementing this project. Besides, the design stages from the project flow chart will be explained further in this section. The tools used in assisting this project have also been defined.

3.1 PROCEDURE IDENTIFICATION

Described in this section is the methodologies applied in order to achieve the final objective set. The methodology used has been illustrated with a project flowchart as shown in Section 3.1.1.

It is important to note that the tasks and workflow for this project is largely based on the Project Gantt Chart. Milestones have been set accordingly and the Gantt chart will be used as a guide along the duration of the project. It is important to note that the Gantt chart will be revised along the course of the project to suit the personal needs of the author as well as to cater for some unforeseen circumstances. Please refer to **Appendix 1** for the Project Gantt Chart.

3.1.1 Project Flow Chart

Figure 3.1 is a flow chart that illustrates the design process used in the implementation of this project.

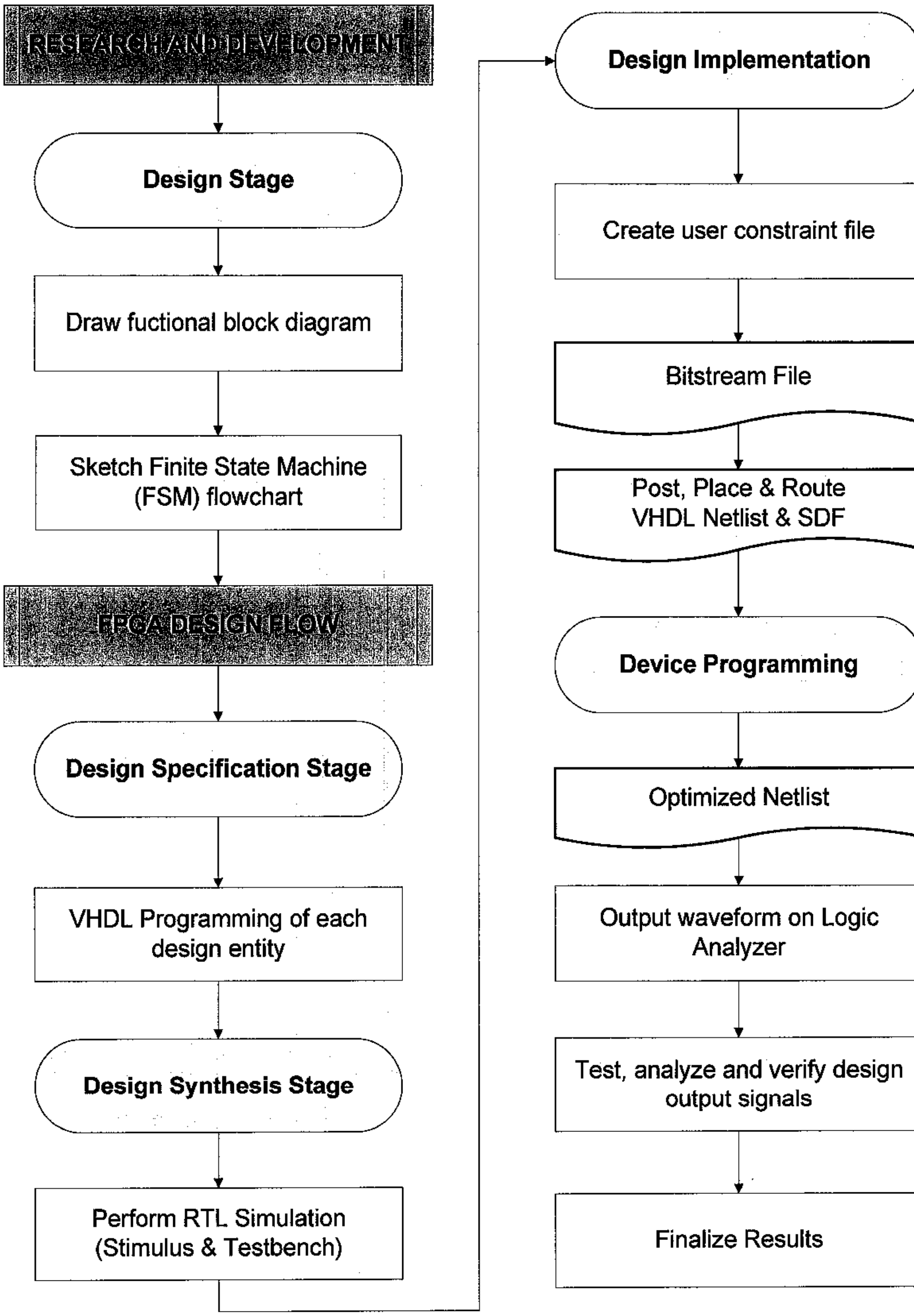


Figure 3.1: Project Flow Chart

From Figure 3.1, it is observed that the methodology has been divided into two major phases which is Research and Development and FPGA Design Flow. The first phase consists of only one sub-stage which is the Design stage. The Design stage will be explained in Section 3.2.1. The FPGA Design Flow phase is a step-by-step method employed to implement the CAN design in FPGA chip. This phase consists of four sub-stages namely the Design Specification stage, Design Synthesis stage, Design Implementation stage and Device Programming stage which will be elaborated in Section 3.3.1, 3.3.2, 3.3.3 and 3.3.4 respectively.

The Research and Development phase includes preliminary research work on CAN from resources like books and internet as well as mastering the VHDL programming language. In semester one, the author completed the first phase and a section of the second phase which is until the Design Synthesis stage. The second semester is a continuation of work from the first semester until completion. In order to achieve the device programming stage, a systematic approach has been employed in order to achieve the final objective. The FPGA design flow has been adopted in order to be able to successfully program the CAN design into the FPGA chip.

3.2 RESEARCH AND DEVELOPMENT

3.2.1 Design Stage

3.2.1.1 Functional Block Diagram

The specification stage involves producing a Functional Block Diagram of a CAN controller with message arbitration capabilities. Figure 3.2 illustrates the block diagram for a CAN controller. From Figure 3.2, it is shown that four main modules are needed to design a CAN system. Enclosed within the double line box are three different modules or entities used to design a single CAN controller, say CAN controller A. The modules are a shift register controller, a shift register and a comparator which is basically an XNOR gate. Outside the double line box is another shift register, a dummy shift register which functions as another CAN controller, say CAN controller B which will only shift out a sequence of bits every clock cycle but will not posses the arbitration properties of a real CAN controller. CAN controller B will compete in the use of the bus with CAN controller A. An AND gate which acts as the design physical bus is part of the FPGA design implementation to demonstrate the message handling capability of the controller across the bus which behave according to the "wired-AND" mechanism as discussed in section 2.5.1.

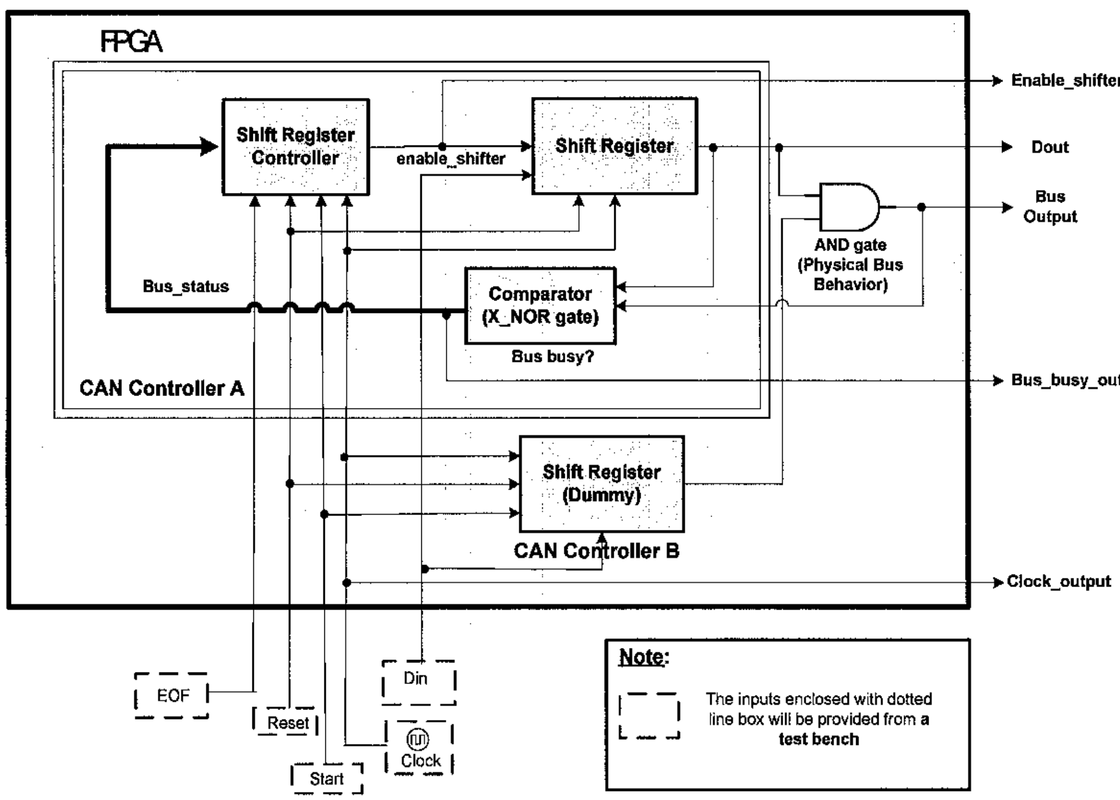


Figure 3.2: Functional Block Diagram for CAN controller

3.2.1.2 CAN Message Handling Design System

In the CAN system, CAN controller A and CAN controller B must send out its identifier value to the bus first to determine its priority. As being mentioned in Section 2.5, CAN adopts a message-based protocol and priority of message is determined by its identifier. The lower the identifier value, the higher the priority. As such, any CAN controller with the lower identifier value will win the arbitration process (message handling process) and thus be able to proceed in sending out its message (the whole data frame) to the receiver across the bus. In this design, CAN controller A will be set to have a higher identifier value than CAN controller B. As such, for this system, CAN controller B will win in the arbitration process as it has been given higher priority due to its lower identifier value as detected by the network system bus. This arbitration process protocol must be achieved to verify the functionality of the CAN message handling system. A diagram which illustrates the CAN message handling system has been shown Figure 3.3.

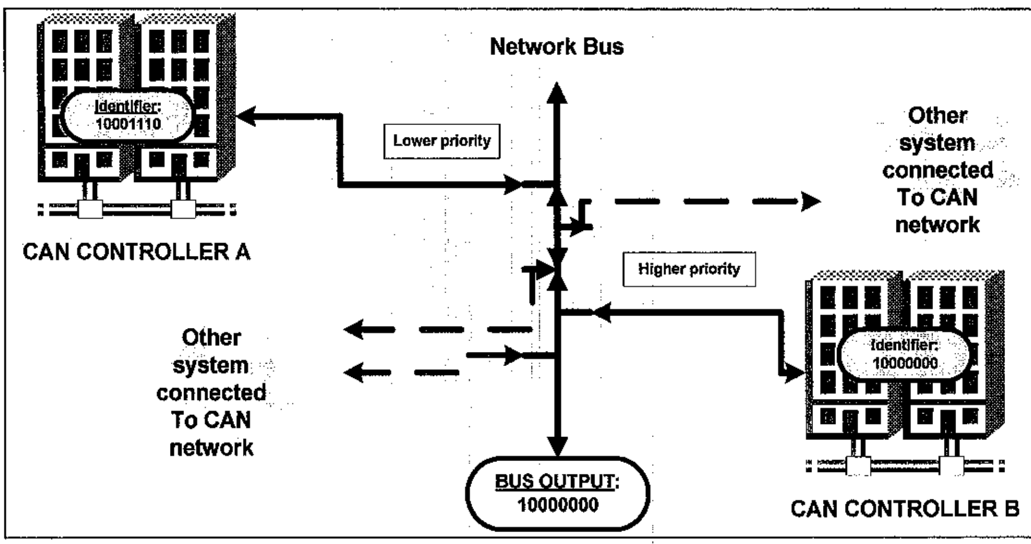


Figure 3.3: A CAN message handling system

3.2.1.3 Finite State Machine (FSM) Chart

Finite State Machine (FSM) chart that describes the shift register controller and shift register in accordance to the functional block diagram are then drawn to assist in the HDL programming stage. A Finite State Machine flowchart leads directly to a hardware realization using VHDL. Basically, the VHDL description of these systems is constructed from the FSM Chart and the VHDL codes are then simulated (RTL behavioral simulation described in Chapter 4) to verify its correct operation. The FSM charts of both the shift register controller and shift register are shown in Figure 3.4 and Figure 3.5 respectively.

Finite State Machine Chart For Shift Register Controller

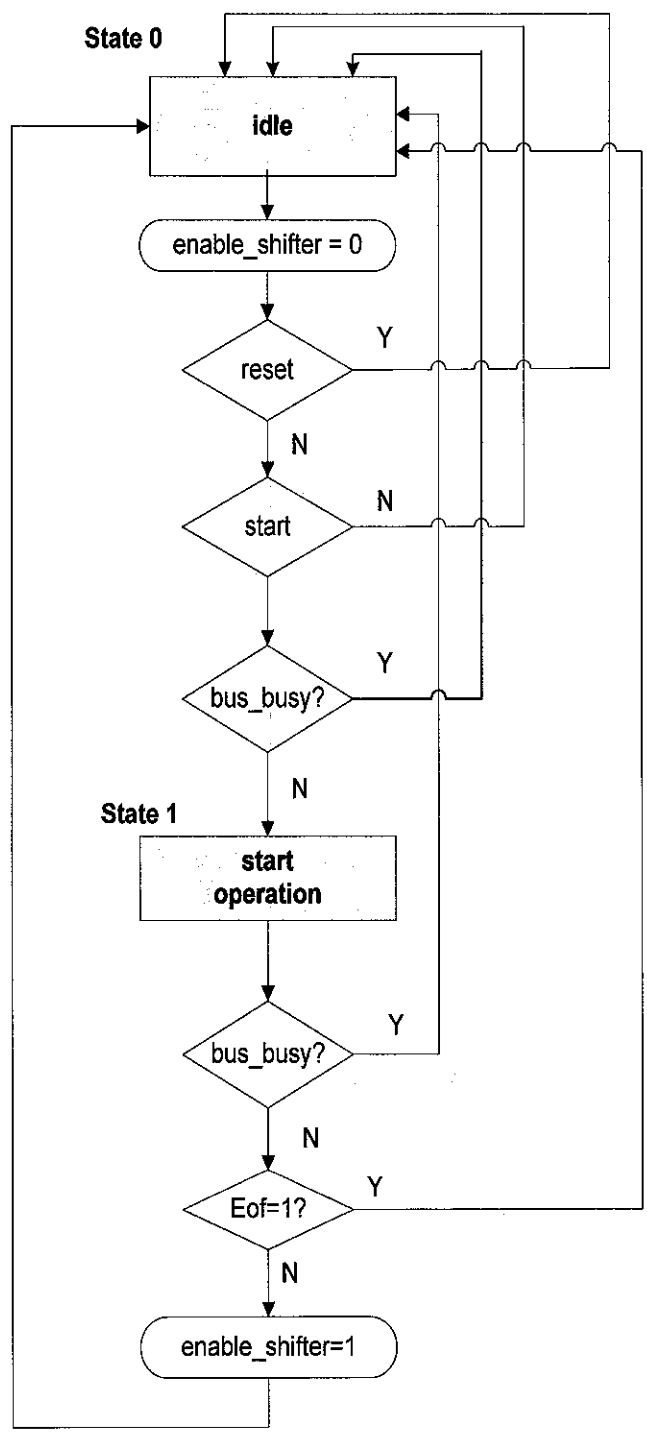


Figure 3.4: Finite State Machine Chart for Shift Register Controller

Finite State Machine Chart for Shift Register

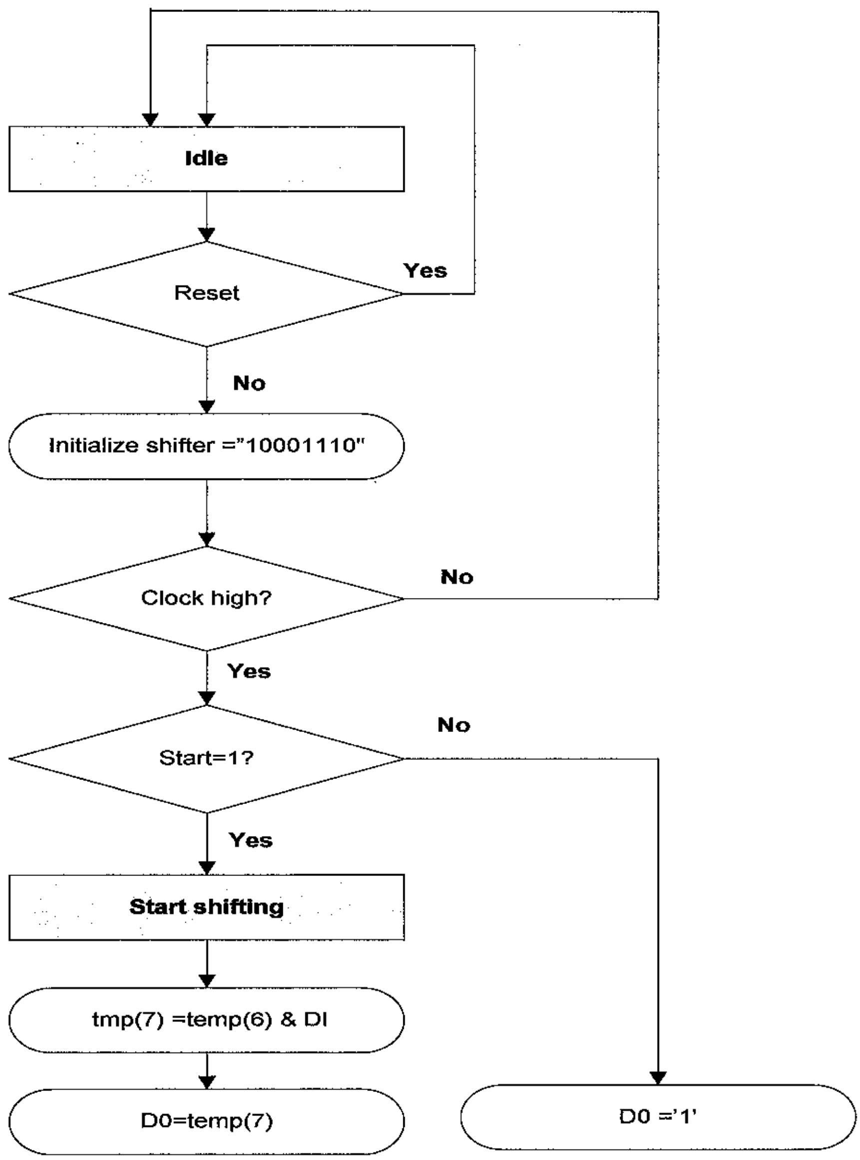


Figure 3.5: Finite State Machine Chart for 8-bit Serial-in, Serial-out Shift Register

From Figure 3.4, it can be observed that the shift register controller has only two states. The minimal number of stages used ensures a more efficient approach to handle the controller. This is because the state of the design is synchronous and relies on the system clock. With less state changes when the design is triggered, the results can be observed immediately. This is an important criterion as the design is a time-critical design according to Mealy state machine. As such, the outputs are a function of the inputs and the current state. Hence, with fewer states, state transition can be designed to happen immediately in the current clock cycle instead of changing only during the next clock cycle. This is an important protocol as the message handling process of each controller must be quick in response. Any failure to do that will disrupt the message sending process.

From Figure 3.5, the shift register module is initially idle. At this idle state, its output (D0) is set to be logic '1' to signify that it is idle. It is designed to send out a sequence of bits after reset is initiated and its start input is activated. The bits will be shifted out serially according to its initialization bits. From the figure, the initialization bits are set as "10001110". After the first eight bits being shifted out, the follow-up bit will be in accordance to the state of the shifter input, DI. The similar process repeats after a reset.

3.3 FPGA DESIGN FLOW

The general FPGA design flow diagram employed is shown in Figure 3.6. This is the overall development methodology used in implementing the CAN design in FPGA.

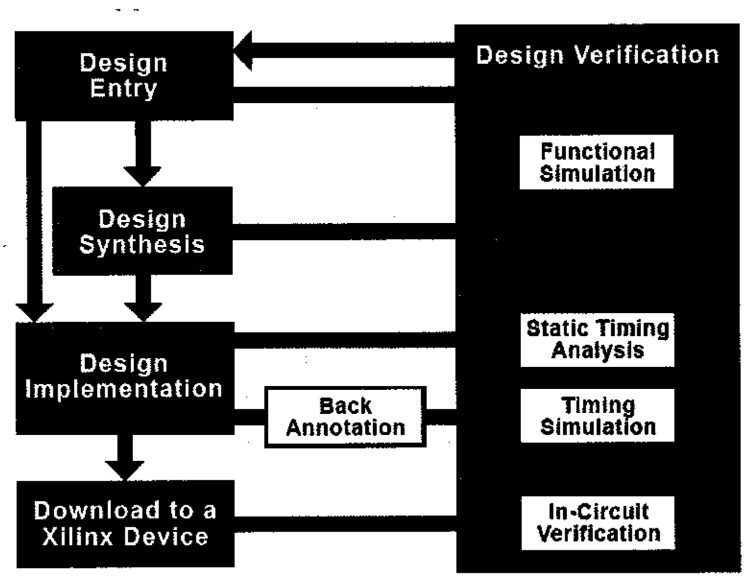


Figure 3.6: FPGA Design Flow [14]

3.3.1 Design Specification Stage

After the first phase, intensive coding with VHDL language is done in accordance to the FSM chart produced earlier in Section 3.2.1. Active-HDL 5.1 program is used as the authoring platform. Simpler module like the XNOR gate code is obtained from web resources and being modified accordingly to suit the needs and specification of the design. Block diagram is used to interconnect the smaller module of the design to produce the top-level module which can be automatically generated by the Active-HDL program. The top-level module is used to tie all other modules to form a complete design of the CAN controller. Please refer to **Appendix 2** for the VHDL source codes for each entity/module and **Appendix 3** for the Top-level module block diagram.

3.3.2 Design Synthesis Stage

Synthesis is the transformation of an idea into a manufacturability device to carry out an intended function. It other words, it can also be described as the transformation of a design from abstract to concrete design [14]. Synthesis will be done using Active-HDL 5.1 and Xilinx Synthesis Technology (XST) program packaged within the ISE Design Environment 4.2i.

The source code for the Comparator, Shift registers, Shift register controller and top-level CAN controller entities will be compiled and synthesized using the Active-HDL program and later migrated to the Xilinx ISE Design Environment 4.2i tool to be synthesized again. Simulation is performed on each entity to ensure that the design works according to specification. As such, Register Transfer Level (RTL) simulation is done to determine and analyze the functionality of the design and to verify the correctness of the RTL VHDL description. The simulated output can also be used to measure the performance of the design and further improvement on the design can be done to improve its performance. The results and the corresponding discussion on the RTL simulation carried out in this project will be presented in Chapter 4.

3.3.3 Design Implementation Stage

Design implementation stage begins with the mapping of a logical design file to a specified device and is complete when the physical design has been successfully routed and a bitstream is generated [14]. Design implementation is also done using the ISE Design Environment 4.2i. The software uses the following design flow engine to carry out the implementation stage.

i. **Translate** - Merge all input netlist to form a complete full chip netlist. This is done by running the *NGDbuild* program.
ii. **Map** - Optimizes the merged netlist by NGDbuild. This can be accomplished by running the program, *MAP*.
iii. **Place & Route** - All logic blocks are assigned specified location within the die. Routing (connection) of logical blocks are done by the program, *PAR*.
iv. **Configure** - Configures the physical implementation into binary stream. This is accomplished by the program *BitGen*. *PromGen* program will then converts *BitGen* into PROM file format.
v. **Timing** - Performs timing analysis by *TRACE* program.

Before an implementation, constraints must first be set. Constraints are instructions placed on symbols or nets in an FPGA schematic or textual entry file such as VHDL or Verilog. They can indicate a number of things such as placement, implementation, naming, signal direction, and timing considerations.

In the Xilinx development system, logical constraints are placed in a file called the User Constraints File. The Xilinx Constraints Editor which is integrated within the ISE Design Environment software is used to create and modify timing and physical constraints of the design. Input files to the Constraints Editor are the UCF file. Constraints created by the user are written to this file and NGD (Native Generic Database) file. This file serves as input to the mapper, which generates the physical design database (NCD file). *NGDBuild* uses the UCF file and design source netlists to produce an NGD file. The NGD is read by the *MAP* program, which generates an

NCD file (a physical design database) and a PCF (Physical Constraints File). The implementation tools use the NCD and PCF files to produce a bitstream. The UCF file can be viewed from **Appendix 13**.

3.3.4 Device Programming Stage

Device programming is the process of loading a design-specific programming into one or more FPGAs in order to define the functional operation of the internal blocks as well as their interconnections. The Xilinx device which will be used for this project is re-programmable and it also supports in-system programming. Device programming is done using the *iMPACT* program within the ISE Design Environment.

The *iMPACT* configuration tool is a command line and GUI based tool, which allows user to configure FPGA designs using Boundary-Scan, Slave Serial, and Select Map configuration modes. Boundary-Scan mode is an industry standard serial programming mode and will be the selected mode to perform the design. External logic from a cable, microprocessor, or other device is used to drive the JTAG specific pins, Test Data In (TDI), Test Mode Select (TMS), and Test Clock (TCK) and sense device response on Test Data Out (TDO). This mode is the most popular mode of configuration due to its standardization and ability to program FPGAs, PLDs, and PROMs through the same four JTAG pins. [14]

There is a specific order in which commands must be executed using the *iMPACT* tool. The following steps are performed to initiate the device programming process:

- i. Set the configuration mode
- ii. Set up the cable port
- iii. Define the JTAG chain and assign files
- iv. Program the device
- v. Verify the device
- vi. Exit from the programming software

The programmed device will be verified by checking the output signals of the board using a Logic Analyzer. The results will be analyzed and discussed in Chapter 4.

3.4 TOOLS USED

The software required to assist in the implementation of this project are the Active-HDL 5.1 and Xilinx ISE 4.2i Design Environment software which is used to perform the steps in Section 3.2 and Section 3.3.

The FPGA board that used in the project is the **Virtex II XC2V1000-FG256** demo board by Insights Electronics Inc, distributed by Memec Design. The Xilinx XC2V1000 FPGA chip used in the project is mounted on the Xilinx FPGA demo board. The FPGA chip on the board contains as much as one million logic gates. The board utilizes the Xilinx XC18V04 ISP PROM, which allows user to download revisions of a design and verify the design changes in order to meet the final system-level design requirements. In addition to ISP PROM, the board also provides a JTAG connector for direct configuration of the Virtex II FPGA. The graphical picture as well as the reference board block diagram of the Xilinx Virtex II demo board is shown in **Appendix 14**.

The output signals from the FPGA chip will be analyzed using a **Hewlett Packard (HP) 1673G** Series Logic Analyzer.

CHAPTER 4

RESULTS AND DISCUSSION

In this section, the RTL simulation results for all the design modules are shown. The waveforms obtained are then analyzed to check if the results are as desired and whether it conformed to the design specifications and requirements. Besides, the results of design implementation and device programming have also been presented in this section. The final design output from the FPGA captured with the Logic Analyzer is being compared with the RTL simulation and discussed further.

4.1 DESIGN SIMULATION RESULTS

Two methods of RTL simulation has been employed in verifying the modules of the design. One is simulation using stimulus and the other is simulation with test benches. Some explanation on both methods is provided in Section 4.1.1 and Section 4.1.2. The corresponding simulation results and discussions for both methods are also provided.

4.1.1 RTL simulation using stimulus

This method of simulation is considered manual simulation as the stimulus is set by the designer itself. In the Active-HDL program, the stimulus is set using "HOTKEY" which is any of the keys from the keyboard to represent a signal state. A stimulus or stimulator that represents the design environment is then used to drive the design and check to make sure that the results produced by the design are as expected. A standard VHDL simulator can be used to read the RTL VHDL description and to verify the correctness of the design. The VHDL simulator reads the VHDL description and then compiles it into an internal format which then executes the compiled format using test vectors [12].

By observing the output waveforms from the simulation, the functionality of the design can be verified. The waveform display shows the values of the signals of the design over time. The results of the simulation using stimulus for XNOR gate, shift register and shift register controller and top-level CAN controller are shown in Figure 4.1, Figure 4.2, Figure 4.3 and Figure 4.4 respectively.

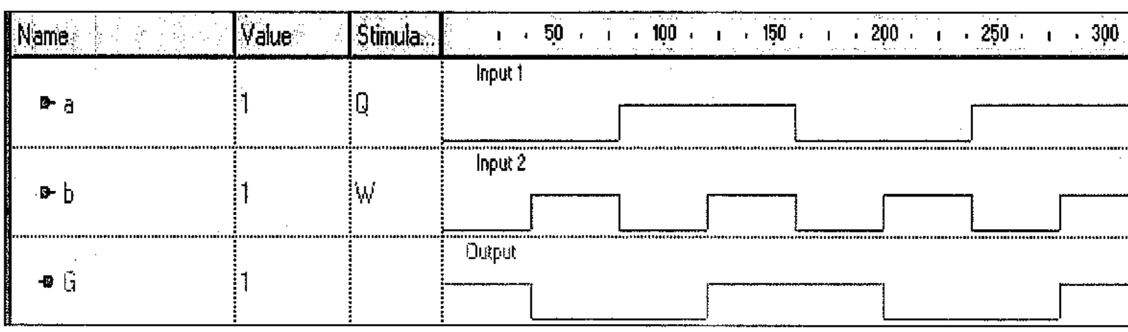


Figure 4.1: RTL simulation using stimulus for XNOR gate

From Figure 4.1, the results obtained is as desired. The output (G) of the XNOR gate is a logic high (logic '1') if both inputs (a and b) is similar (either $a = b =$ '0' or $a = b =$ '1') while the output is logic low (logic '0') if both inputs are not similar. This is the behavior expected from that of an XNOR gate. The XNOR gate is used as a comparator in this CAN design to compare the signals transmitted and received again from controller A to check if it has been arbitrated or not. The comparator will compare the output signals obtained from the CAN controller before and after its output passed through the AND gate. The AND gate is used to emulate a real physical bus which have the characteristic of an AND gate when it carries messages.

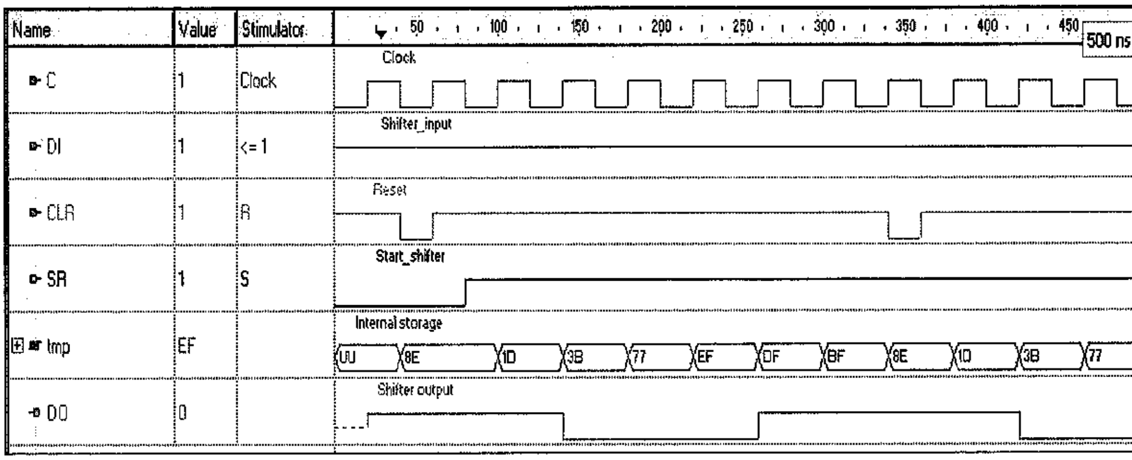


Figure 4.2: RTL simulation using stimulus for 8-bit shift register

The shift register is an 8-bit serial-in, serial-out shift register and it is set to have input initialization bits of "10001110". The shift register will shift the bits at each clock event (clock high). From Figure 4.2, the clock frequency is set at 25 MHz to emulate the clock frequency of the FPGA on-board oscillator. *CLR* represents a reset and the design must be reset before it is activated. As soon as the shifter is initiated by starting up the shifter (*SR* ='1'), it is observed that the first eight bits of the shifter output (*D0*) is "10001110", which is the initialization bits of the shift register. The ninth, tenth eleventh and so forth bits will be similar to that of the shifter input value (*DI*) which is "1" until the shifter is reset again. The shifter input (*DI*) allows real time input into the shifter. From the simulation, it is shown that during the second reset, the output will again be similar to the input initialization bits as the shifter is being reset after the eleventh bit. This is the case as after reset, the shifter will be restored with the eight initialization bits again.

The RTL simulation for the dummy shift register module will not be shown as it has similar characteristic to that of this shift register. The only slight difference is in its initialization bits output. The dummy shift register is set to have an initialization bit of "100000000". And as such, it will shift out the initialization bits every clock cycle. Similarly the shifter input (*DI*) has been set at '1'.

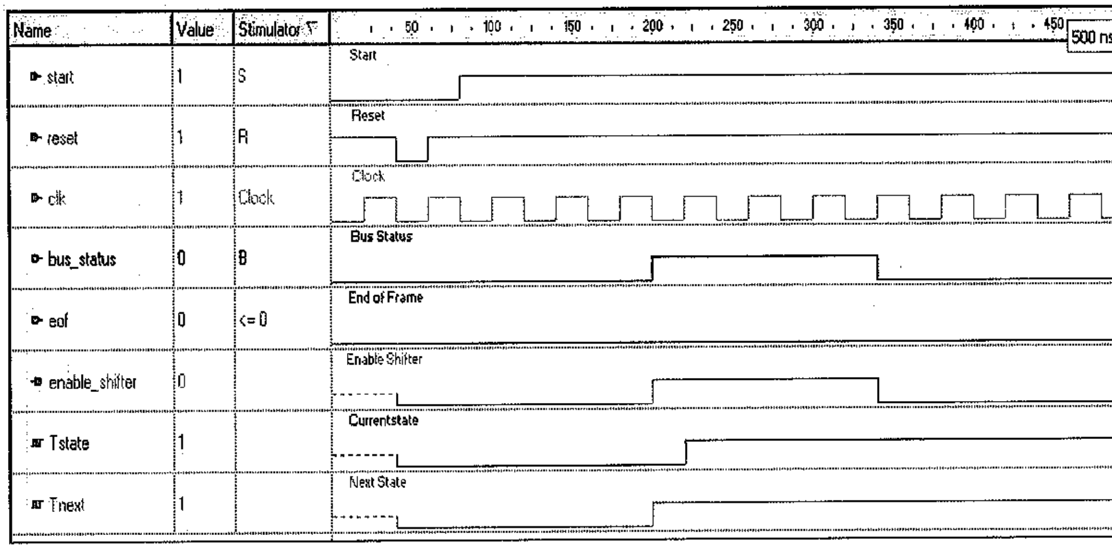


Figure 4.3: RTL simulation using stimulus for shift register controller

The clock frequency is set at 25 MHz. The value 25 MHz is chosen as the on-board oscillator of the FPGA demo board is approximately this frequency range. The input signals are *start, reset, clk, eof* and *bus status* while the output signals are *enable_shifter*. The *Tstate* and *Tnext* are the internal signals which represent the states of the design.

From Figure 4.3, it is observed that the shift register controller is activated when the signal is fed into its '*start*' input. The *bus_status* signal as it names implies indicates whether the bus is free or busy. A logic '1' represents the bus is free while logic '0' represents the bus is busy. The shift register controller will output a logic '1' signal (*enable_shifter* = '1') whenever the bus status is not busy and vice versa. This is the signal that will be used to enable or disable the shift register.

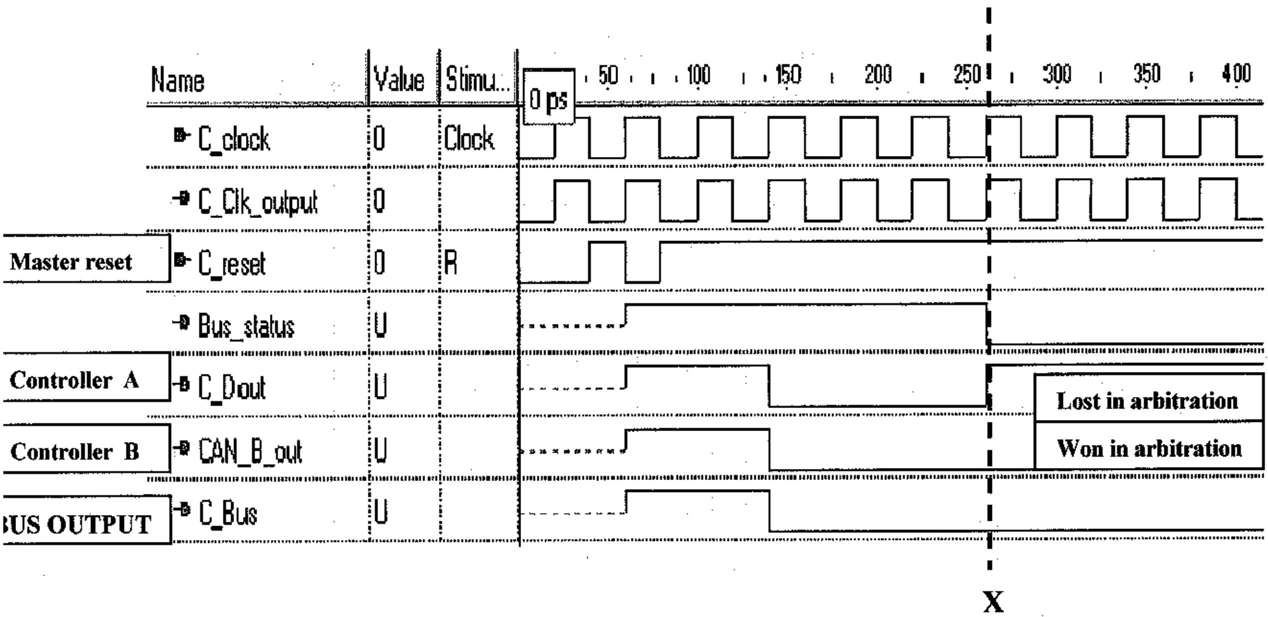


Figure 4.4: RTL simulation using stimulus for top-level CAN controller

Referring to Figure 4.4, *C_bus* is the output from the AND gate, which in this case, acts as a physical bus which carries the messages transmitted by the transmitter (Controller A and Controller B) to the receiver. *C_Dout* is the output from CAN controller A, the main CAN controller which exhibits the arbitration characteristics. *C_clock* is the clock input which has been set at 25 MHz. *C_Clk_output* is the clock output. The reason the clock output is checked is to ensure that the clock goes into the design during the design implementation stage. *C_reset* is the master reset of the system. Before the start of the message sending process, the C_reset input must be set to low (Logic '0') to reset the whole system. *Bus_status* is an output which represents the status of the bus whether the bus is free or busy. Comparing the waveform of *C_bus*, *C_Dout* and *C_CAN_B_out*, it was found that the waveform for *C_bus* and *C_CAN_B_out* is similar. Hence, the bus is actually carrying the sequence of bits sent by CAN Controller B. This shows that CAN controller B has actually won in the arbitration process. CAN controller A has lost in the arbitration process at **X** (please refer to Figure 4.4) because it has a higher identifier value as compared to CAN controller B. This means that controller B actually has a higher priority than controller A and is given the bus allocation.

The results obtained indicate that the design has met with the specification of a CAN system during message handling. The arbitration of signals has been exhibited by the controller when it lost in the bus allocation due to its lower priority identifier.

4.1.2 RTL simulation using Test Bench

A test bench is a design entity which serves as a host environment for another design being tested. Test bench is not real device or a system that must communicate with its environment and as such it does not need any inputs or outputs. The tested entity is called Unit Under Test (UUT) and it is instantiated in the test bench architecture. The ports of the UUT instantiation will be assigned stimuli signals by the test bench architecture. The heart of each test bench is a set of stimuli which is a sequence of values for each UUT input signal applied over time. Since test bench does not communicate with its environment through signals, all stimuli must be declared internally in the test bench architecture like any other signals inside the VHDL architecture declarative part. Test vectors used to simulate the UUT entity can be furnished in an external file or encoded immediately in the test bench architecture [10].

The advantage of using test bench is the fact that once test bench is generated as well as its test vectors are specified, it can be reused many times to perform simulation and automatic verification of our design regardless of any successive revisions of the VHDL designs. The predicted outputs can also be coded into the test bench. As such, the test bench not only prepares the test vectors but can verify the expected output from the design. As such, the outputs can be check once the test bench is run and the outcome or results for the simulation can be reported. Report clause is used in the test bench to display messages when something goes wrong or if the simulation is not successful. The report of simulation can be viewed from the console window of the VHDL program.

Due to constant revisions being done on the design entities, test benches are written for the shift register module, shift register controller module as well as the top-level CAN controller module to verify their functionality. The results of the test bench simulations are shown in Figure 4.5, Figure 4.6, Figure 4.7 and Figure 4.8. The results of the test bench can be viewed from the *ERR_STATUS* (error status) output. Besides, a report will be generated on the console window by the Active-HDL program to indicate the successful simulation status. An example of the generated report for the top-level CAN controller module is shown in Figure 4.9.The test benches source codes can be viewed at **Appendix 4**.

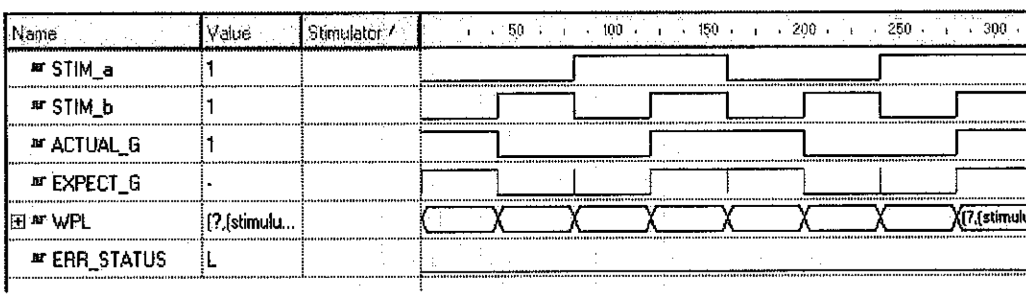


Figure 4.5: Test bench simulated output for XNOR gate.

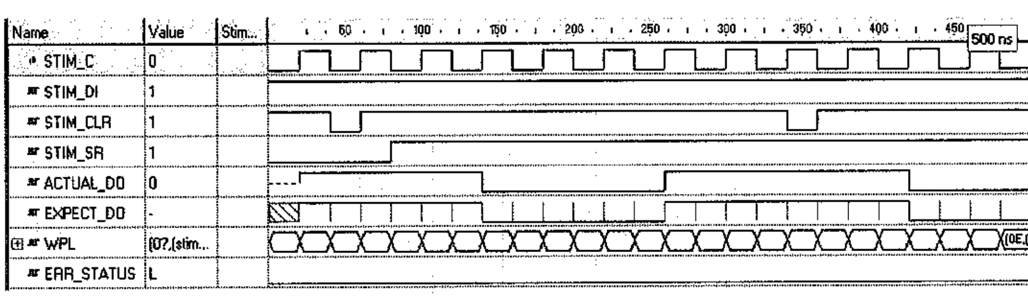


Figure 4.6: Test bench simulated output for 8-bit shift register.

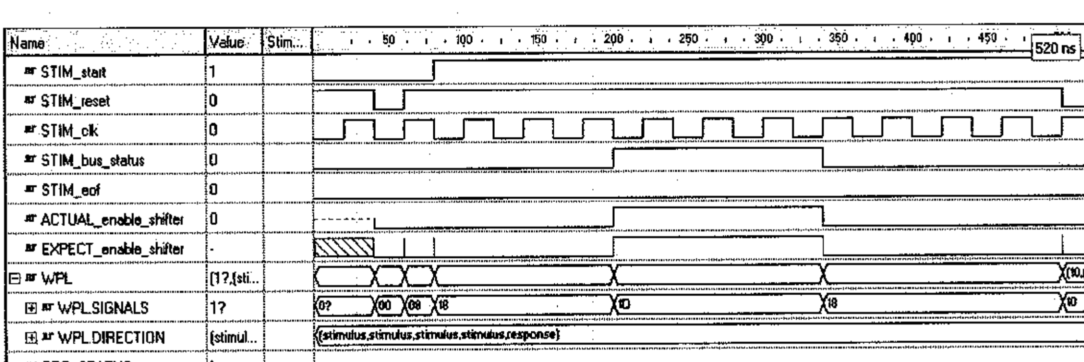


Figure 4.7: Test bench simulated output shift register controller.

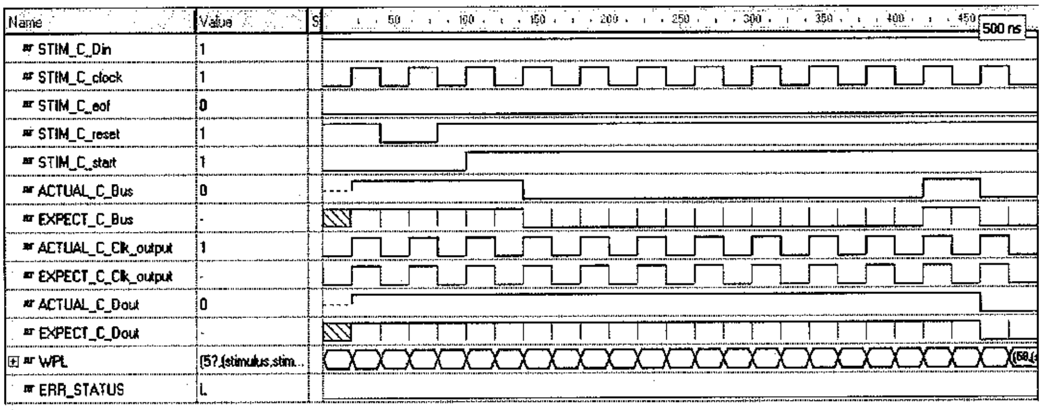


Figure 4.8: Test bench simulated output for top-level CAN controller

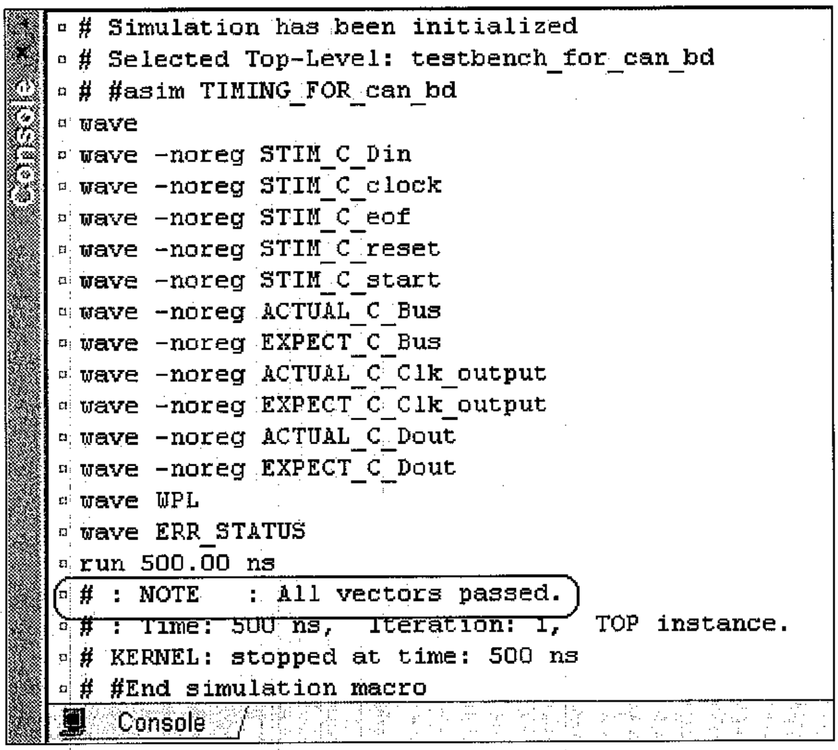


Figure 4.9: Report of Top-level CAN controller simulation on window console

The observations from the output waveforms verified that the design in Figure 4.5, Figure 4.6, Figure 4.7 and Figure 4.8 is functioning as desired as the *ERR_STATUS* which denotes the error status of the simulated module shows logic '0'. Logic '0' proves that the simulated module is correct and has no errors in syntax and its hierarchy. Figure 4.9 shows a console generated report that verifies that the top-level module has been successfully simulated. It is important to note that the successful results obtained is not as spontaneous and simple as it may seem. The modules has been revised, debugged and re-tested many times before the desired results can be achieved.

Basically, the outputs expected from the test benches is in essence similar to that obtained through RTL stimulus simulation. Test bench has the benefits of reusability whereby user will not need to supply the stimulus each time simulation is performed as the test vectors has been written beforehand.

4.2 DESIGN SYNTHESIS AND IMPLEMENTATION RESULTS

As being mentioned earlier in Chapter 3, the design synthesis and implementation stage has been done using the ISE Design Environment 4.2i software. Synthesis is performed using the XST (Xilinx Synthesis Technology) software while implementation has been done with the help of multiple software tools like Xilinx FloorPlanner and FPGA Editor which comes packaged within the ISE Design Environment.

During implementation, the design is converted from the logical design file format created in the design entry stage into a physical file format contained in an NCD (Native Circuit Description) file. Implementation processing for FPGAs involves three basic phases: Translate, Map, and Place and Route as described in Section 3.3.3. Processes to check and verify timing requirements are also included. At the end of these phases, a programming file can be created. With the programming file, user can directly download the programming file into the Xilinx device.

The completed implementation of the design will generate the following reports which provide a complete description of the FPGA based design.

4.2.1 Translation Report

During the first step of design implementation, the translate process merges all of the input netlists and design constraint information and outputs a Xilinx NGD (Native

Generic Database) file. The output NGD file can then be mapped to the targeted device family. The Translation Report contains warning and error messages from the three translation processes which are conversion of the EDIF or XNF style netlist to the Xilinx NGD netlist format, timing specification checks, and logical design rule checks. All errors must be rectified before the implementation can be preceded. Please refer to **Appendix 5** for the Translation Report.

4.2.2 Map report

The MAP process first performs a logical DRC (Design Rule Check) on the design in the NGD file produced by the Translate process. MAP then maps the logic to the components (logic cells, I/O cells, and other components) in the target Xilinx FPGA. The output design is an NCD (Native Circuit Description) file physically representing the design mapped to the components in the Xilinx FPGA. The NCD file can then be placed and routed.

The MAP report contains warning and error messages detailing logic optimization and problems in mapping logic to physical resources. Basically, the report provides a detailed description of the design information and design summary after the design is mapped onto the FPGA.

Some important information gathered from this report is the number of gate count required for the design. The number of gate count for this design is only 171 gates. The target architecture used for this project, the Xilinx XC2V100 chip can support up to one million gates and as such is more than enough to support the CAN controller design needs.

The Map report also includes the following information; Removed logic summary, IOB properties and Area Group Summary and Modular Design summary. The Map report can be viewed from **Appendix 6**.

4.2.3 Place & Route Report

After an FPGA design has undergone the necessary processing to bring it into the mapped NCD format, it is ready to be placed and routed. This phase is done by PAR (Xilinx's Place and Route program). PAR takes a mapped NCD file, places and routes the design, and produces an NCD file to be used by the programming file generator (BitGen). The output NCD file can also act as a guide file if the place and route the design is repeated again due to some minor changes done on the design.

The Place & Route report contains routing information or connection of logical blocks within the FPGA hardware. The report also contains the device utilization summary, the delay summary and the average connection delay summary. The average connection delay summary highlights the maximum pin delay of the design and the listing of each pin delays in nanoseconds. Please refer to **Appendix 7** for the Place & Route Report.

It is important to note that the FPGA device is actually a gate-array-like architecture, with a matrix of logic cells surrounded by periphery of Input/Output (I/O) cells. Segments of metal interconnect are linked in an arbitrary fashion by programmable switches in order to form the desired signal nets between the cells. The CAN design which have been mapped and downloaded into the FPGA device will combine an abundance or combination of logic gates ,registers and I/Os to form the design interconnection.

The logic signals generated in the block of FPGA are called the Control Logic Block (CLB). In addition to CLBs, the FPGA has programmable input/output blocks (I/O blocks) located within the chip. Flip-flops and buffers are also located within the FPGA. The placement of the gates, flip flops and buffers in the FPGA cab be reviewed and edited after the Pace & Route step using the FPGA Floor Planner tool from the ISE Design Environment like Floorplanner and FPGA Editor. The Floorplanner displays a hierarchical representation of the design using hierarchy structure lines and colors to distinguish the different hierarchical levels. The complete connection of the design in the Xilinx XC2V1000 FPGA chip can be viewed from **Appendix 8.**

4.2.4 Pad Report

The Pad report contains I/O pin information that is a list of the pin-out by pin name and list of pin-out by pin number. The Pad report is important for future maintenance, expansion and troubleshooting of the design as it contains the critical pin information of the design. Please refer to **Appendix 9** for the Pad Report.

4.2.5 Asynchronous Delay Report

This report highlights the delay analysis of all the nets and connections of the design. Each signal nets is analyzed and then tabulated. The twenty worst net delays has been tabulated in the report and this information is important and must not be taken lightly as time delays will affect the performance of time-critical design. The propagation delay in the design can be improved by focusing on the nets with the worst delay. Please refer to **Appendix 10** for the Asynchronous Delay Report.

4.2.6 Post-Place & Route Static Timing Report

The Post-Place & Route Static Timing Report process contains a calculated worst-case timing for all signal paths of a design. It optionally includes a complete listing of all delays on each individual path in the design. This report also tabulated a checklist of all timing constraints in the design. It is important to check and verify that all timing constraints are met in the implementation of the design. The Post-Place & Route Static Timing report can be viewed from **Appendix 11**.

4.2.7 Programming File Generation Report

After the design has been completely routed, the device is configured so that it can execute the desired function. Xilinx's bitstream generation program, BitGen, takes a fully routed NCD (Native Circuit Description) file as its input and produces a configuration bitstream (a binary file with a .bit extension). The BIT file contains all of the configuration information from the NCD file defining the internal logic and interconnections of the FPGA, plus device-specific information from other files associated with the target device. The binary data in the BIT file can then be downloaded into the FPGA's memory cells, or it can be used to create a PROM file.

The Programming File Generation Report or also known as the BitGen Report is the final report generated in the implementation step. The report lists the errors and warnings found during the bit map generation. The bit stream file generated is very crucial as it will be downloaded into the FPGA. The BitGen report can be viewed from **Appendix 12**.

4.3 DEVICE PROGRAMMING RESULTS

The device programming stage proved successful as there is no error generated during the FPGA chip programming process is initiated until it has completed. After the FPGA chip has been programmed, the output signals are analyzed with a logic analyzer to verify its correct operation.

4.3.1 Logic Analyzer Output Waveform

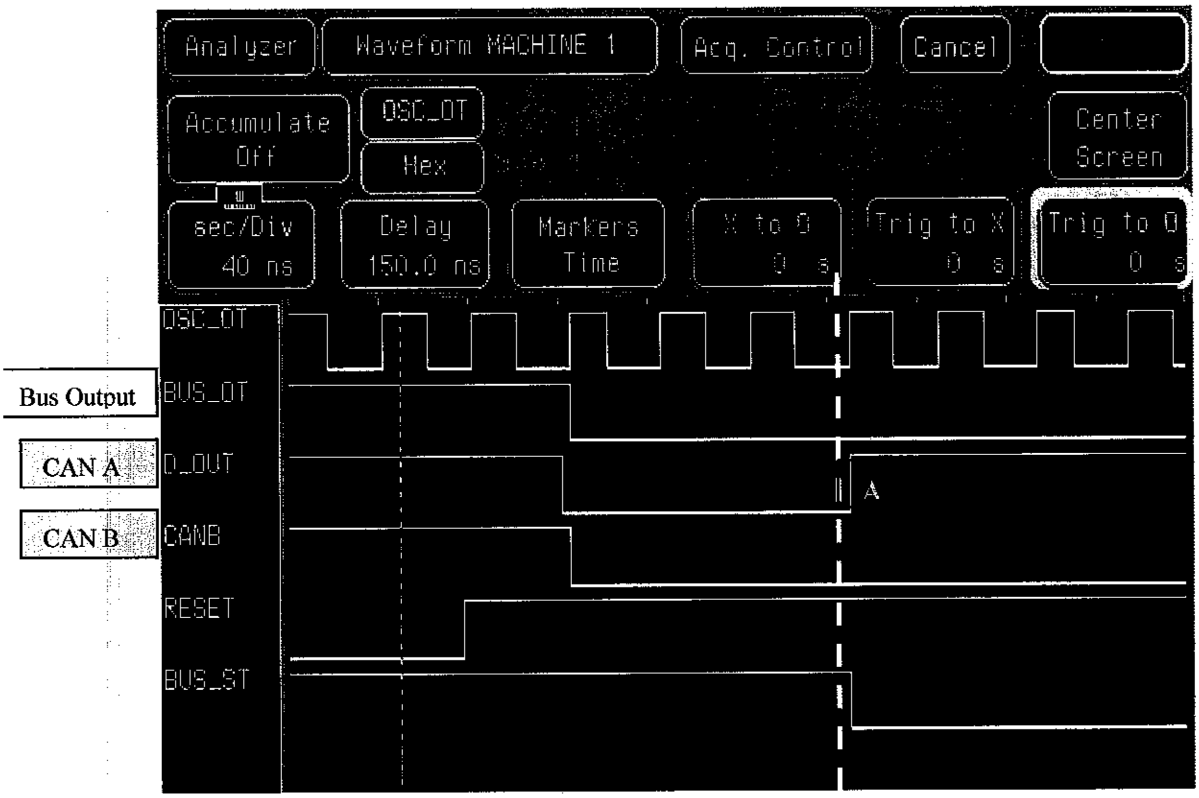


Figure 4.10: CAN top-level Logic Analyzer output waveform

From Figure 4.10, it is observed that the signals captured from the Logic Analyzer are almost similar to the RTL simulation results in Figure 4.4. Note that the signals *OSC_OT* denotes the oscillator output signals, *BUS_OT* denotes the bus output signals, *D_OUT* is the output signal of CAN controller A, *CANB* denotes the output signals from CAN controller B, *RESET* is the master reset signal of the design and *BUS_ST* denotes the status of the bus.

The process starts as soon as the reset signal (*RESET*) is initiated. It is observed that the CAN controller A output (*D_OUT*) lost its arbitration starting from point A (referring to Figure 4.10). The bus status signal (*BUS_ST*) in turn shows a logic '0' which indicates that the bus is busy. CAN controller B (*CANB*) which has lower identifier value won in the arbitration process and thus be able to send its full data frame across the bus. Hence, the bus output (*BUS_OT*) is similar to that of CAN controller B (*CANB*).

The results obtained from the RTL simulation as well as the Logic Analyzer captured waveforms verify that the CAN design is working fine according to the specification set in Chapter 3. However further improvements can be done to the design to improve on its performance and functionality. This will be discussed in further in Chapter 5.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

This section reviews and concludes the project while highlighting some of the problems faced and how it is handled to overcome them. Some recommendations are made to suggest for further improvement and for future progress.

5.1 CONCLUSION

The desired deliverable is a CAN controller that will exhibits bit-level non-destructive arbitration of signals during message collision. From the successful simulation results obtained as well as the captured output waveform from the Logic Analyzer in Chapter 4, the objectives set in Section 1.3 have already been achieved.

This project was carried out in two semesters. The first semester was mainly dedicated to preliminary research work, detailed design of the CAN message handling protocol and also testing and performing RTL simulation of the design. The second semester work mainly focused on the implementation of the design in FPGA.

The mastering of a new language, in this case, VHDL proved to be most challenging part of the project in this semester. As the language is not taught in the university and there is no expertise among the university lecturers and technicians in this field, self-study and self-exploration have to be done to familiarize with the language. The language is different to other common programming languages like C++ or Visual Basic. The knowledge of sequential programming in which the author is familiar with is not sufficient to assist in the concurrent programming environment. There was a need to understand that the operation of a digital system is inherently concurrent and so the VHDL programming techniques must be concurrent as well.

Trial-and-error method is used for familiarization with the authoring tool and language became the norm for many weeks before the author proceeds to intensive coding. With time and effort, the author has managed to grasp the language better and be able to code the modules and achieved the results desired for the design.

The major problem encountered during the second semester was mainly caused by the constraint of time available for the author to familiarize with the FPGA development system. The FPGA Design Flow includes the utilization of two separate software tools and many sub tools embedded within the two main softwares. The main software tools mentioned are the Aldec Active-HDL and Xilinx ISE Design Environment. Each of these software tools has different function and needed to be fully understood before design development could begin. There is also a lack of user friendliness in the software tools and this has resulted in a longer familiarization time taken as compared to development time.

However, upon completion, the project has indeed enhanced the author's understanding in digital design using VHDL. Besides, the author has gain valuable insights on the techniques used in FPGA implementation, particularly each step in the design flow from design specification to device programming stage for FPGA implementation. The author has also gain more knowledge on CAN message handling protocol and its other functionality.

5.2 RECOMMENDATIONS

The CAN controller can be further improved by adding in more functionality like error handling capability in the design. As arbitration of signals signify that the signals sent by a node/station is lost during transmission, an error handling capability may detect the lost transmission and will be able to recover the lost signals by informing the node in particular to resent the message.

Besides, the design can be optimized further by reducing the clock frequencies used in the system and also by optimizing the timing constraints used in the design. The current clock frequency used approximately 24 MHZ. In such frequency, the system may be affected by noise interference and as such may affect the performance of the design. Besides, delay in the design will be larger due to parasitic capacitance. Parasitic capacitance may occur in the routing or wiring within the chip especially for routing between IOBs which is in close proximity. As such, lower clock frequency will be more feasible to prevent delays and interference.

In addition to that, the AND gate used in the CAN design to represent the physical can be replaced with a real physical wire for future project enhancement. The behavior of the CAN design system using the real physical wire can then be compared with the one with an AND gate to verify the feasibility and functionality of the CAN design.

In conclusion, this project has achieve all the objectives set in Section 1.3 which is to be able to deliver an FPGA-based implementation of a version of CAN system with emphasis on bit-level non-destructive arbitration.

REFERENCES

1. Mike J Schofield 14th August 2003
   <**http://www.mjschofield.com/canworks.htm**>
2. Siemens. October 1998
   <**http://wwwlhc.icepp.s.utokyo.ac.jp/ATLAS/tgcelex/technology/can/CANPRES.pdf**>
3. Hitex UK Ltd. 15th August 2003
   <**http://www.hitex.co.uk/CAN/canarticle.html**>
4. Microchip. 16th August 2003
   <**http://www.microchip.com/download/appnote/analog/can/00713a.pdf**>
5. Cia (CAN in Automation). 16th August 2003
   < **http://www.can-cia.de/can/protocol/**>
6. Charles H. Roth, Jr, 1998. *Digital Systems Design Using VHDL*, Boston, PWS Publishing Company
7. Kevin Skahill, 1996. *VHDL for Programmable Logic*, Addison-Wesley Logman, Inc.
8. Douglas Perry, 1998. *VHDL*, New York, McGraw-Hill Companies Inc.
9. Sudhakar Yalamanchili, 2001. *Introductory VHDL: From Simulation to Synthesis*, New Jersey, Prentice-Hall, Inc.
10. J.Mirkowski,MKapustka, Z. Skowronski, A. Biniszkiewicz, 1998. *Active VHDL Series:Book #2*, ALDEC, Inc.
11. Technical report refers BOSCH CAN Specification 2.0. (1991)
12. Thesis refers to Cecilia Chau. (2001)
13. Thesis refers to Abu Bakar Sayuti. (2002)
14. Xilinx Software Manual (2004)

APPENDICES

APPENDIX 1

Project Gantt Chart For Final Year Design Project

Semester 1 & Semester 2

No.	Detail/ Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	**Selection and Confirmation of Project Title**														
2	**Preliminary Research Work**														
	-Read up on CAN														
	-Read up on VHDL Programming														
3	**Submission of Preliminary Report**				●										
4	**Project Exploration and Tutorial**														
	-Explore the Active-HDL Program														
	-Learn VHDL language														
5	**Submission of Progress Report**							●							
6	**Basic Project Implementation**														
	- Design functional block diagram of a CAN controller														
	-Draw the Finite State Machine for each modules used in the block diagram														
	-Program each module														
	-Combine all modules to produce top-level module														
7	**Testing and Debugging Process**														
8	**Design simulation**														
9	**Complete Interim Report**														

LEGEND

● **Suggested milestone**

Process

1.	**Improvement on previous project design**																
2.	**Project Synthesis and Debugging**																
	-Using Project Navigator by Xilinx																
3.	**Progress Report 1 Submission**		●														
4.	**Device Implementation**																
	-Understanding selected FPGA Board architecture																
5.	**Device Programming**																
6.	**Progress Report 2 Submission**						●										
7.	**Troubleshooting FPGA chip**																
	-Identify and debug the errors																
8.	**Investigation of CAN controller chip performance**																
	-Testing and analysis of controller with testing equipment																
9.	**Submission of Draft Report**									●							
10.	**Submission of Final Report/Dissertation**												●				
11.	**Oral Presentation Preparation (Exhibition)**														●		
12.	**Extended Abstract Submission**																●

LEGEND

● Suggested milestone

Process

APPENDIX 2

VHDL Source Codes

- XNOR Gate
- Shift Registers
- Shift Register Controller
- Top-level CAN Controller

__**VHDL source code for XNOR entity**__

```
-------------------------------------------------------------------------
--Design name: can19
--XNOR gate
--Revision 2.0
--Last updated: 23/10/03
-------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;


entity XNOR_ent is
port(   a: in std_logic;
        b: in std_logic;
        G: out std_logic
);
end XNOR_ent;

architecture behv of XNOR_ent is
begin

   G <= a  xnor b;

end behv;
-------------------------------------------------------------------------
```

VHDL source code for shifter entity

```
------------------------------------------------------------------------
--Design name: can40
--8-bit Shift-Left Register
--Revision 2.2
--Last updated: 10/2/04
------------------------------------------------------------------------
--Details:
--Shifter with Positive-Edge Clock, Asynchronous Clear, Serial In, and Serial Out
------------------------------------------------------------------------
--Input Description:
------------------------------------------------------------------------
--C = Positive-Edge Clock
--DI = Serial In
--CLR = Asynchronous Clear (active High)
--DO = Serial Output
--SR=Start Register
------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity shifter is
        port(C,DI,  CLR,SR : in std_logic;
                DO : out std_logic);
end shifter;

architecture archi of shifter is
        signal tmp: std_logic_vector(7 downto 0);

begin
        process (C, CLR)
        begin
                if (CLR = '0') then
                        -- reset shift
                        tmp <= "10001110";
                elsif (C'event and C= '1') then
                        -- +ve edge trigger flop
                        if (SR='1') then
                                tmp <= tmp(6 downto 0) & DI;
                                DO  <= tmp(7);
                        else
                                DO <= '1';
                        end if;
                end if;
```

```
        end process;
end archi;
--description:
--when it is +ve edge of clock and SR is low,
--DO is high.
--When it is +ve edge of clock and SR is high,
--send the predefined value and then tmp[0] is replaced by DI.
--This is asynchronous shift register,
--when CLR is low at any time,
--the shift register will be reset and tmp is '1001110'

-----------------------------------------------------------------------
```

VHDL source code for dummy shifter entity

```
-----------------------------------------------------------------------
--Design name: can27
--8-bit Shift-Left Register (Dummy)
--Revision 1.0
--Last updated: 3/3/04
-----------------------------------------------------------------------
--Details:
--Shifter with Positive-Edge Clock, Asynchronous Clear, Serial In, and Serial Out
-----------------------------------------------------------------------
--Input Description:
-----------------------------------------------------------------------
--C = Positive-Edge Clock
--DI = Serial In
--CLR = Asynchronous Clear (active High)
--DO = Serial Output
--SR=Start Register
-----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity dummy_shifter is
	port(C,DI,  CLR,SR : in std_logic;
		DO : out std_logic);
end dummy_shifter;

architecture arch_shifter of dummy_shifter is
	signal tmp: std_logic_vector(7 downto 0);

begin
	process (C, CLR)
	begin
		if (CLR = '0') then
			-- reset shift
			tmp <= "10000000";
		elsif (C'event and C= '1') then
			-- +ve edge trigger flop
			if (SR='1') then
				tmp <= tmp(6 downto 0) & DI;
				DO  <= tmp(7);
			else
				DO <= '1';
			end if;
		end if;
```

```
        end process;
end arch_shifter;
--description:
--when it is +ve edge of clock and SR is low,
--DO is high.
--When it is +ve edge of clock and SR is high,
--send the predefined value and then tmp[0] is replaced by DI.
--This is asynchronous shift register,
--when CLR is low at any time,
--the shift register will be reset and tmp is '1001110'

-------------------------------------------------------------------
```

VHDL source codes for shift register controller entity

```
-----------------------------------------------------------------------
--Design name: can40
--Shift register controller
--Revision 2.2
--Last updated: 10/2/04
-----------------------------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
-----------------------------------------------------------------------
entity SR_controller is
        port (reset,start,clk,eof,bus_status:in STD_LOGIC;
                enable_shifter:out STD_LOGIC);
end SR_controller;
-----------------------------------------------------------------------
architecture arbitration of SR_controller is
        signal Tstate,Tnext: STD_LOGIC;

begin
        process(clk, reset)
        begin
                if ( reset = '0') then
                        Tstate <= '0';  -- make reset as asynchronous so whenever reset is
high, the controller will be set to 00 state
                else
                        if ( clk'event and clk ='1' ) then
                                Tstate <= Tnext;
                        else
                                Tstate <= Tstate;
                        end if;
                end if;

        end process;

        process(start,bus_status,eof,Tstate,reset)

        begin
                case Tstate is
                        when '0' =>
                        if (reset='0') then
                                enable_shifter <= '0';
                                Tnext <= '0' ;
                        elsif (start='1' and bus_status = '1' ) then -- if bus is free
                                enable_shifter <='1';
                                Tnext<= '1';
```

```
                    else
                              enable_shifter <= '0';
                              Tnext <= '0';
                    end if;


                    when '1' => ---
                    if (start = '1') then
                              if (bus_status ='1')  then          -- if bus is free
                                        if (eof = '0') then  -- if end of frame is not reached
                                                  enable_shifter <='1';
                                                  Tnext<='1';
                                        else
                                                  enable_shifter <= '0';  --if bus is not free
                                                  Tnext <= '0';
                                        end if;
                              else
                                        enable_shifter <= '0'; -- it is started but the bus is
busy so wait here.
                                        Tnext<='1';
                              end if;
                    else
                              enable_shifter <= '0';
                              Tnext <= '0';     -- the start is low
                    end if;

                    when others=>null;
          end case;

     end process;
end arbitration;


-- Description
-- one is reset state one is transmission state

---------------------------------------------------------------------------
```

VHDL source codes for can top-level entity

```
-------------------------------------------------------------------------------
--
-- Title       : can_bd
-- Design      : can54
-- Author      : Lai Yeen
-- Company     : UTP
--
-------------------------------------------------------------------------------
--
-- File        : C:\My_Designs\can54\compile\can_bd.vhd
-- Generated   : Tue Apr  6 14:50:06 2004
-- From        : C:/My_Designs/can54/src/can_bd.bde
-- By          : Bde2Vhdl ver. 2.01
--
-------------------------------------------------------------------------------
--
-- Description :
--
-------------------------------------------------------------------------------
-- Design unit header --
library IEEE;
use IEEE.std_logic_1164.all;


entity can_bd is
  port(
       C_clock : in STD_LOGIC;
       C_reset : in STD_LOGIC;
       C_Bus : out STD_LOGIC;
       C_Clk_output : out STD_LOGIC;
       C_Dout : out STD_LOGIC;
       Reset : out STD_LOGIC
  );
end can_bd;

architecture can_bd of can_bd is

---- Component declarations -----

component dummy_shifter
  port (
       C : in STD_LOGIC;
       CLR : in STD_LOGIC;
```

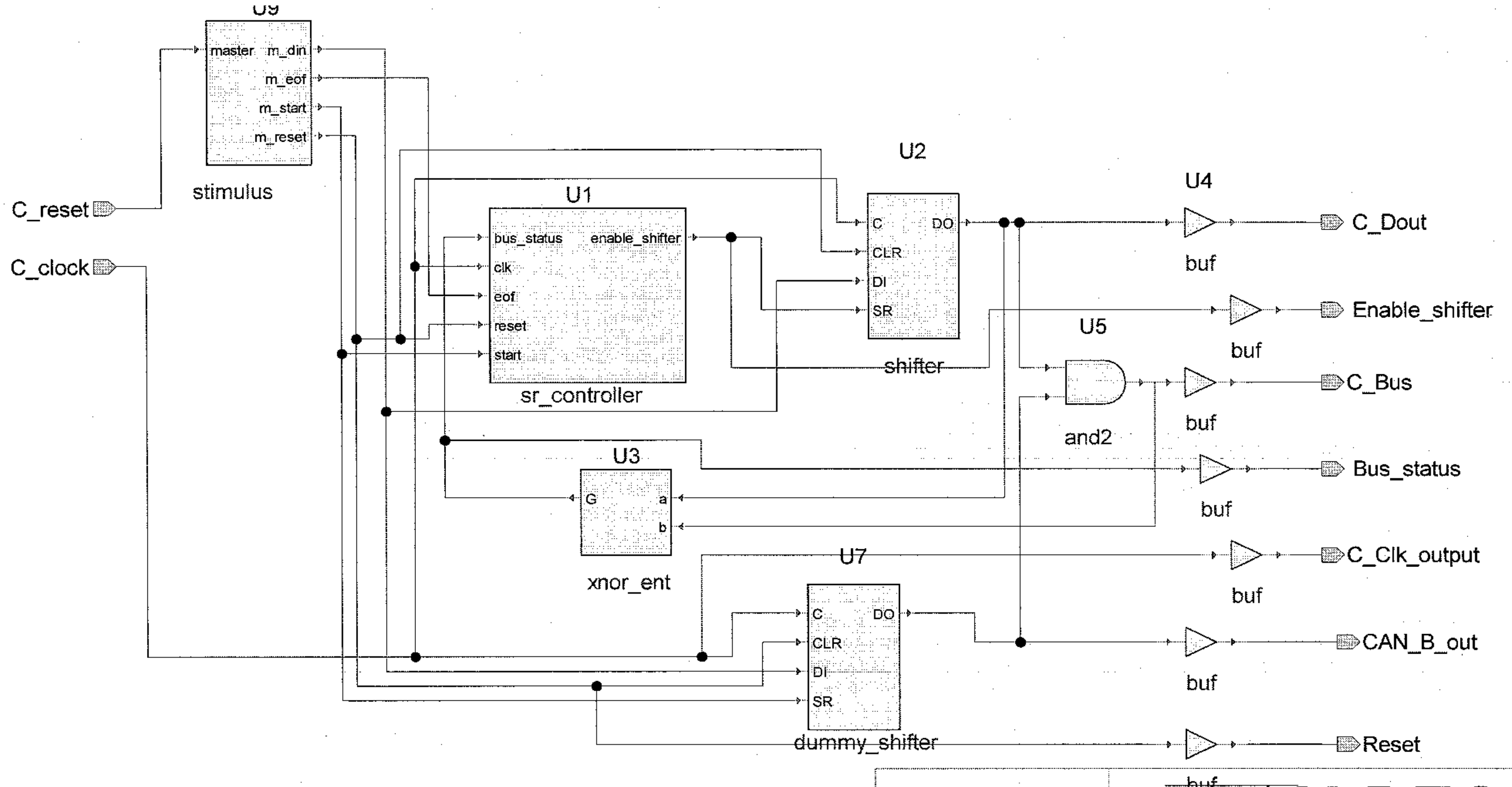
```
    DI : in STD_LOGIC;
    SR : in STD_LOGIC;
    DO : out STD_LOGIC
 );
end component;
component shifter
 port (
    C : in STD_LOGIC;
    CLR : in STD_LOGIC;
    DI : in STD_LOGIC;
    SR : in STD_LOGIC;
    DO : out STD_LOGIC
 );
end component;
component sr_controller
 port (
    bus_status : in STD_LOGIC;
    clk : in STD_LOGIC;
    eof : in STD_LOGIC;
    reset : in STD_LOGIC;
    start : in STD_LOGIC;
    enable_shifter : out STD_LOGIC
 );
end component;
component stimulus
 port (
    master : in STD_LOGIC;
    m_din : out STD_LOGIC;
    m_eof : out STD_LOGIC;
    m_start : out STD_LOGIC
 );
end component;
component xnor_ent
 port (
    a : in STD_LOGIC;
    b : in STD_LOGIC;
    G : out STD_LOGIC
 );
end component;

---- Signal declarations used on the diagram ----

signal NET10903 : STD_LOGIC;
signal NET1151 : STD_LOGIC;
signal NET4277 : STD_LOGIC;
signal NET4389 : STD_LOGIC;
```

```
signal NET5095 : STD_LOGIC;
signal NET5127 : STD_LOGIC;
signal NET5136 : STD_LOGIC;
signal NET98 : STD_LOGIC;

begin

---- Component instantiations ----

U1 : sr_controller
 port map(
     bus_status => NET1151,
     clk => C_clock,
     enable_shifter => NET4277,
     eof => NET98,
     reset => C_reset,
     start => NET5136
 );

Reset <= C_reset;

U2 : shifter
 port map(
     C => C_clock,
     CLR => C_reset,
     DI => NET5127,
     DO => NET4389,
     SR => NET4277
 );

U3 : xnor_ent
 port map(
     G => NET1151,
     a => NET4389,
     b => NET10903
 );

C_Dout <= NET4389;

NET10903 <= NET5095 and NET4389;

C_Bus <= NET10903;

U7 : dummy_shifter
 port map(
     C => C_clock,
```

```
    CLR => C_reset,
    DI => NET5127,
    DO => NET5095,
    SR => NET5136
 );

C_Clk_output <= C_clock;

U9 : stimulus
 port map(
    m_din => NET5127,
    m_eof => NET98,
    m_start => NET5136,
    master => C_reset
 );


end can_bd;
```

APPENDIX 3

Block Diagram of Top-level CAN Controller



(C)ALDEC. Inc 2230 Corporate Circle Henderson, NV 89074	ALDEC The Design Verification Company
Created:	4/8/04
Title:	can_bd

APPENDIX 4

Test Benches for RTL Simulation

- XNOR Gate
- Shift Register
- Shift Register Controller
- Top-level CAN Controller

Test Bench for XNOR entity

```
-------------------------------------------------------------------------------
--
-- Title      : CAN
-- Design     : can54
-- Author     : Lai Yeen
-- Company    : UTP
--
-------------------------------------------------------------------------------
--
-- File       : xnor_entwb_TB.vhd
-- Generated  : Sun Apr  4 16:55:11 2004
-- From       : xnor_entwb_TB_settings.txt
-- By         : tb_generator.pl ver. ver 1.2s
--
-------------------------------------------------------------------------------
--
-- Description : main Test Bench entity
--
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

use IEEE.waves_interface.all;
use WORK.UUT_test_pins.all;
use WORK.waves_objects.all;
use WORK.DESIGN_DECLARATIONS.all;
use WORK.MONITOR_UTILITIES.all;
use WORK.WAVES_GENERATOR.all;

-- User can put library and packages declaration here

entity xnor_ent_wb is
end xnor_ent_wb;

architecture xnor_entwb_archi of xnor_ent_wb is

        -- Component declaration of the tested unit
        component xnor_ent

        port (
                a : in std_logic;
                b : in std_logic;
                G : out std_logic);
```

```
        end component;

        -- Internal signals declarations:
        --   stimulus signals (STIM_)for the waveforms mapped into UUT inputs,
        --   observed signals (ACTUAL_) used in monitoring ACTUAL Values of UUT
Outputs,
        --   bi-directional signals (BI_DIRECT_) mapped into UUT Inout ports,
        --   the BI_DIRECT_ signals are used as stimulus and also used for monitoring
the UUT Inout ports
        signal STIM_a : std_logic;
        signal STIM_b : std_logic;
        signal ACTUAL_G : std_logic;

        -- Expected signals used in monitoring the UUT OUTPUTS
        signal EXPECT_G : STD_ULOGIC;
        -- WAVES signals OUTPUTing each slice of the waves port list
        signal WPL  : WAVES_PORT_LIST;
        signal TAG  : WAVES_TAG;
        signal ERR_STATUS: STD_LOGIC:='L';
        -- Signal END_SIM denotes end of test vectors file
        signal END_SIM : BOOLEAN:=FALSE;

begin

        -- Process that generates the WAVES waveform
        WAVES: WAVEFORM (WPL, TAG);


        -- Processes that convert the WPL values to 1164 Logic Values
        ASSIGN_STIM_a: STIM_a <= WPL.SIGNALS(TEST_PINS'pos(a)+1);
        ASSIGN_STIM_b: STIM_b <= WPL.SIGNALS(TEST_PINS'pos(b)+1);
        ASSIGN_EXPECT_G: EXPECT_G <= WPL.SIGNALS(TEST_PINS'pos(G)+1);


        -- Unit Under Test port map
        UUT: xnor_ent
        port map(
                a => STIM_a,
                b => STIM_b,
                G => ACTUAL_G);
        -- Monitor processes to verify the UUT operational response
        MONITOR_G:

        MONITOR_RESULTS(REP_FILE,ACTUAL_G,EXPECT_G,NOW,G_NAME,E
RR_STATUS);
```

```
        -- Process denoting end of test vectors file
        NOTIFY_END_VECTORS: process (TAG)
        begin
                if TAG.len /= 0 then
                        if ERR_STATUS='L' then
                                report "All vectors passed.";
                        elsif ERR_STATUS='1' then
                                report "Errors were encountered on the output ports,
differences are listed in xnor_ent_report.log";
                        end if;
                        END_SIM <= TRUE;
                        CLOSE_VECTOR;
                        CLOSE_REPORT;
                end if;
        end process;

end xnor_entwb_archi;


configuration TESTBENCH_FOR_xnor_ent of xnor_ent_wb is
        for xnor_entwb_archi
                for UUT : xnor_ent
                        use entity work.xnor_ent (behv);
                end for;
        end for;
end TESTBENCH_FOR_xnor_ent;
```

Test Bench for shifter entity

```
-------------------------------------------------------------------------------------------------
--
-- Title      : CAN
-- Design     : can54
-- Author     : Lai Yeen
-- Company    :UTP
--
-------------------------------------------------------------------------------------------------
--
-- File       : shifterwb_TB.vhd
-- Generated  : Sun Apr  4 16:44:33 2004
-- From       : shifterwb_TB_settings.txt
-- By         : tb_generator.pl ver. ver 1.2s
--
-------------------------------------------------------------------------------------------------
--
-- Description : main Test Bench entity
--
-------------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

use IEEE.waves_interface.all;
use WORK.UUT_test_pins.all;
use WORK.waves_objects.all;
use WORK.DESIGN_DECLARATIONS.all;
use WORK.MONITOR_UTILITIES.all;
use WORK.WAVES_GENERATOR.all;

-- User can put library and packages declaration here

entity shifter_wb is
end shifter_wb;

architecture shifterwb_archi of shifter_wb is

	-- Component declaration of the tested unit
	component shifter

	port (
		C : in std_logic;
		DI : in std_logic;
		CLR : in std_logic;
```

```
            SR : in std_logic;
            DO : out std_logic);
        end component;

        -- Internal signals declarations:
        --  stimulus signals (STIM_)for the waveforms mapped into UUT inputs,
        --  observed signals (ACTUAL_) used in monitoring ACTUAL Values of UUT
Outputs,
        --  bi-directional signals (BI_DIRECT_) mapped into UUT Inout ports,
        --   the BI_DIRECT_ signals are used as stimulus and also used for monitoring
the UUT Inout ports
        signal STIM_C : std_logic;
        signal STIM_DI : std_logic;
        signal STIM_CLR : std_logic;
        signal STIM_SR : std_logic;
        signal ACTUAL_DO : std_logic;

        -- Expected signals used in monitoring the UUT OUTPUTS
        signal EXPECT_DO : STD_ULOGIC;
        -- WAVES signals OUTPUTing each slice of the waves port list
        signal WPL  : WAVES_PORT_LIST;
        signal TAG  : WAVES_TAG;
        signal ERR_STATUS: STD_LOGIC:='L';
        -- Signal END_SIM denotes end of test vectors file
        signal END_SIM : BOOLEAN:=FALSE;

begin

        -- Process that generates the WAVES waveform
        WAVES: WAVEFORM (WPL, TAG);


        -- Processes that convert the WPL values to 1164 Logic Values
        ASSIGN_STIM_C: STIM_C <= WPL.SIGNALS(TEST_PINS'pos(C)+1);
        ASSIGN_STIM_DI: STIM_DI <= WPL.SIGNALS(TEST_PINS'pos(DI)+1);
        ASSIGN_STIM_CLR: STIM_CLR <=
WPL.SIGNALS(TEST_PINS'pos(CLR)+1);
        ASSIGN_STIM_SR: STIM_SR <= WPL.SIGNALS(TEST_PINS'pos(SR)+1);
        ASSIGN_EXPECT_DO: EXPECT_DO <=
WPL.SIGNALS(TEST_PINS'pos(DO)+1);


        -- Unit Under Test port map
        UUT: shifter
        port map(
            C => STIM_C,
```

```
            DI => STIM_DI,
            CLR => STIM_CLR,
            SR => STIM_SR,
            DO => ACTUAL_DO);
        -- Monitor processes to verify the UUT operational response
        MONITOR_DO:

        MONITOR_RESULTS(REP_FILE,ACTUAL_DO,EXPECT_DO,NOW,DO_NA
ME,ERR_STATUS);

        -- Process denoting end of test vectors file
        NOTIFY_END_VECTORS: process (TAG)
        begin
                if TAG.len /= 0 then
                        if ERR_STATUS='L' then
                                report "All vectors passed.";
                        elsif ERR_STATUS='1' then
                                report "Errors were encountered on the output ports,
differences are listed in shifter_report.log";
                        end if;
                        END_SIM <= TRUE;
                        CLOSE_VECTOR;
                        CLOSE_REPORT;
                end if;
        end process;

end shifterwb_archi;


configuration TESTBENCH_FOR_shifter of shifter_wb is
        for shifterwb_archi
                for UUT : shifter
                        use entity work.shifter (archi);
                end for;
        end for;
end TESTBENCH_FOR_shifter;
```

Test Bench for shift register entity

```
-------------------------------------------------------------------------------
--
-- Title      : CAN
-- Design     : can54
-- Author     : Lai Yeen
-- Company    : UTP
--
-------------------------------------------------------------------------------
--
-- File       : sr_controllerwb_TB.vhd
-- Generated  : Sun Apr  4 16:57:29 2004
-- From       : sr_controllerwb_TB_settings.txt
-- By         : tb_generator.pl ver. ver 1.2s
--
-------------------------------------------------------------------------------
--
-- Description : main Test Bench entity
--
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

use IEEE.waves_interface.all;
use WORK.UUT_test_pins.all;
use WORK.waves_objects.all;
use WORK.DESIGN_DECLARATIONS.all;
use WORK.MONITOR_UTILITIES.all;
use WORK.WAVES_GENERATOR.all;

-- User can put library and packages declaration here

entity sr_controller_wb is
end sr_controller_wb;

architecture sr_controllerwb_archi of sr_controller_wb is

        -- Component declaration of the tested unit
        component sr_controller

        port (
                start : in std_logic;
                reset : in std_logic;
                clk : in std_logic;
```

```
            bus_status : in std_logic;
            eof : in std_logic;
            enable_shifter : out std_logic);
    end component;

    -- Internal signals declarations:
    --  stimulus signals (STIM_)for the waveforms mapped into UUT inputs,
    --  observed signals (ACTUAL_) used in monitoring ACTUAL Values of UUT
Outputs,
    --  bi-directional signals (BI_DIRECT_) mapped into UUT Inout ports,
    --   the BI_DIRECT_ signals are used as stimulus and also used for monitoring
the UUT Inout ports
    signal STIM_start : std_logic;
    signal STIM_reset : std_logic;
    signal STIM_clk : std_logic;
    signal TMP_clk : std_logic;
    signal STIM_bus_status : std_logic;
    signal STIM_eof : std_logic;
    signal ACTUAL_enable_shifter : std_logic;

    -- Expected signals used in monitoring the UUT OUTPUTS
    signal EXPECT_enable_shifter : STD_ULOGIC;
    -- WAVES signals OUTPUTing each slice of the waves port list
    signal WPL  : WAVES_PORT_LIST;
    signal TAG  : WAVES_TAG;
    signal ERR_STATUS: STD_LOGIC:='L';
    -- Signal END_SIM denotes end of test vectors file
    signal END_SIM : BOOLEAN:=FALSE;

begin

    -- Process that generates the WAVES waveform
    WAVES: WAVEFORM (WPL, TAG);

    CLOCK_GEN_FOR_clk: process
    begin
        if END_SIM = FALSE then
            TMP_clk <= '0';
            wait for 20 ns;
        else
            wait;
        end if;
        if END_SIM = FALSE then
            TMP_clk <= '1';
            wait for 20 ns;
        else
```

```
                    wait;
            end if;
        end process;
        -- Processes that convert the WPL values to 1164 Logic Values
        ASSIGN_STIM_start: STIM_start <=
WPL.SIGNALS(TEST_PINS'pos(start)+1);
        ASSIGN_STIM_reset: STIM_reset <=
WPL.SIGNALS(TEST_PINS'pos(reset)+1);
        ASSIGN_STIM_clk: STIM_clk <= TMP_clk;
        ASSIGN_STIM_bus_status: STIM_bus_status <=
WPL.SIGNALS(TEST_PINS'pos(bus_status)+1);
        ASSIGN_STIM_eof: STIM_eof <= WPL.SIGNALS(TEST_PINS'pos(eof)+1);
        ASSIGN_EXPECT_enable_shifter: EXPECT_enable_shifter <=
WPL.SIGNALS(TEST_PINS'pos(enable_shifter)+1);


        -- Unit Under Test port map
        UUT: sr_controller
        port map(
                start => STIM_start,
                reset => STIM_reset,
                clk => STIM_clk,
                bus_status => STIM_bus_status,
                eof => STIM_eof,
                enable_shifter => ACTUAL_enable_shifter);
        -- Monitor processes to verify the UUT operational response
        MONITOR_enable_shifter:

        MONITOR_RESULTS(REP_FILE,ACTUAL_enable_shifter,EXPECT_enable_s
hifter,NOW,enable_shifter_NAME,ERR_STATUS);

        -- Process denoting end of test vectors file
        NOTIFY_END_VECTORS: process (TAG)
        begin
            if TAG.len /= 0 then
                if ERR_STATUS='L' then
                    report "All vectors passed.";
                elsif ERR_STATUS='1' then
                    report "Errors were encountered on the output ports,
differences are listed in sr_controller_report.log";
                end if;
                END_SIM <= TRUE;
                CLOSE_VECTOR;
                CLOSE_REPORT;
            end if;
        end process;
```

```
end sr_controllerwb_archi;

configuration TESTBENCH_FOR_sr_controller of sr_controller_wb is
	for sr_controllerwb_archi
		for UUT : sr_controller
			use entity work.sr_controller (arbitration);
		end for;
	end for;
end TESTBENCH_FOR_sr_controller;
```

Test Bench for can top-level entity

```
-----------------------------------------------------------------------------------------------
--
-- Title      : CAN
-- Design      : can54
-- Author      : Lai Yeen
-- Company     : UTP
--
-----------------------------------------------------------------------------------------------
--
-- File       : can_bdwb_TB.vhd
-- Generated  : Sun Apr  4 17:03:08 2004
-- From        : can_bdwb_TB_settings.txt
-- By         : tb_generator.pl ver. ver 1.2s
--
-----------------------------------------------------------------------------------------------
--
-- Description : main Test Bench entity
--
-----------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

use IEEE.waves_interface.all;
use WORK.UUT_test_pins.all;
use WORK.waves_objects.all;
use WORK.DESIGN_DECLARATIONS.all;
use WORK.MONITOR_UTILITIES.all;
use WORK.WAVES_GENERATOR.all;

-- User can put library and packages declaration here

entity can_bd_wb is
end can_bd_wb;

architecture can_bdwb_archi of can_bd_wb is

        -- Component declaration of the tested unit
        component can_bd

        port (
                C_Din : in std_logic;
                C_clock : in std_logic;
                C_eof : in std_logic;
```

```
            C_reset : in std_logic;
            C_start : in std_logic;
            C_Bus : out std_logic;
            C_Clk_output : out std_logic;
            C_Dout : out std_logic);
        end component;

        -- Internal signals declarations:
        --  stimulus signals (STIM_)for the waveforms mapped into UUT inputs,
        --  observed signals (ACTUAL_) used in monitoring ACTUAL Values of UUT
Outputs,
        --  bi-directional signals (BI_DIRECT_) mapped into UUT Inout ports,
        --   the BI_DIRECT_ signals are used as stimulus and also used for monitoring
the UUT Inout ports
        signal STIM_C_Din : std_logic;
        signal STIM_C_clock : std_logic;
        signal TMP_C_clock : std_logic;
        signal STIM_C_eof : std_logic;
        signal STIM_C_reset : std_logic;
        signal STIM_C_start : std_logic;
        signal ACTUAL_C_Bus : std_logic;
        signal ACTUAL_C_Clk_output : std_logic;
        signal ACTUAL_C_Dout : std_logic;

        -- Expected signals used in monitoring the UUT OUTPUTS
        signal EXPECT_C_Bus : STD_ULOGIC;
        signal EXPECT_C_Clk_output : STD_ULOGIC;
        signal EXPECT_C_Dout : STD_ULOGIC;
        -- WAVES signals OUTPUTing each slice of the waves port list
        signal WPL  : WAVES_PORT_LIST;
        signal TAG  : WAVES_TAG;
        signal ERR_STATUS: STD_LOGIC:='L';
        -- Signal END_SIM denotes end of test vectors file
        signal END_SIM : BOOLEAN:=FALSE;

begin

        -- Process that generates the WAVES waveform
        WAVES: WAVEFORM (WPL, TAG);

        CLOCK_GEN_FOR_C_clock: process
        begin
            if END_SIM = FALSE then
                TMP_C_clock <= '0';
                wait for 20 ns;
            else
```

```
                wait;
            end if;
            if END_SIM = FALSE then
                TMP_C_clock <= '1';
                wait for 20 ns;
            else
                wait;
            end if;
        end process;
        -- Processes that convert the WPL values to 1164 Logic Values
        ASSIGN_STIM_C_Din: STIM_C_Din <=
WPL.SIGNALS(TEST_PINS'pos(C_Din)+1);
        ASSIGN_STIM_C_clock: STIM_C_clock <= TMP_C_clock;
        ASSIGN_STIM_C_eof: STIM_C_eof <=
WPL.SIGNALS(TEST_PINS'pos(C_eof)+1);
        ASSIGN_STIM_C_reset: STIM_C_reset <=
WPL.SIGNALS(TEST_PINS'pos(C_reset)+1);
        ASSIGN_STIM_C_start: STIM_C_start <=
WPL.SIGNALS(TEST_PINS'pos(C_start)+1);
        ASSIGN_EXPECT_C_Bus: EXPECT_C_Bus <=
WPL.SIGNALS(TEST_PINS'pos(C_Bus)+1);
        ASSIGN_EXPECT_C_Clk_output: EXPECT_C_Clk_output <=
WPL.SIGNALS(TEST_PINS'pos(C_Clk_output)+1);
        ASSIGN_EXPECT_C_Dout: EXPECT_C_Dout <=
WPL.SIGNALS(TEST_PINS'pos(C_Dout)+1);


        -- Unit Under Test port map
        UUT: can_bd
        port map(
                C_Din => STIM_C_Din,
                C_clock => STIM_C_clock,
                C_eof => STIM_C_eof,
                C_reset => STIM_C_reset,
                C_start => STIM_C_start,
                C_Bus => ACTUAL_C_Bus,
                C_Clk_output => ACTUAL_C_Clk_output,
                C_Dout => ACTUAL_C_Dout);
        -- Monitor processes to verify the UUT operational response
        MONITOR_C_Bus:

        MONITOR_RESULTS(REP_FILE,ACTUAL_C_Bus,EXPECT_C_Bus,NOW,C
_Bus_NAME,ERR_STATUS);
        MONITOR_C_Clk_output:
```

```
        MONITOR_RESULTS(REP_FILE,ACTUAL_C_Clk_output,EXPECT_C_Clk_o
utput,NOW,C_Clk_output_NAME,ERR_STATUS);
        MONITOR_C_Dout:

        MONITOR_RESULTS(REP_FILE,ACTUAL_C_Dout,EXPECT_C_Dout,NOW,
C_Dout_NAME,ERR_STATUS);

        -- Process denoting end of test vectors file
        NOTIFY_END_VECTORS: process (TAG)
        begin
                if TAG.len /= 0 then
                        if ERR_STATUS='L' then
                                report "All vectors passed.";
                        elsif ERR_STATUS='1' then
                                report "Errors were encountered on the output ports,
differences are listed in can_bd_report.log";
                        end if;
                        END_SIM <= TRUE;
                        CLOSE_VECTOR;
                        CLOSE_REPORT;
                end if;
        end process;

end can_bdwb_archi;


configuration TESTBENCH_FOR_can_bd of can_bd_wb is
        for can_bdwb_archi
                for UUT : can_bd
                        use entity work.can_bd (can_bd);
                end for;
        end for;
end TESTBENCH_FOR_can_bd;
```

APPENDIX 5

Translation Report

Translation report

Release 4.2i - ngdbuild E.35

Command Line: ngdbuild -dd c:/kly/can54/_ngo -nt timestamp -p xc2v1000-fg256-4

can_bd.ngc can_bd.ngd

Reading NGO file "C:/kly/can54/can_bd.ngc" ...
Reading component libraries for design expansion...

Annotating constraints to design from file "can_bd.ucf" ...

Checking timing specifications ...
Checking expanded design ...

NGDBUILD Design Results Summary:
 Number of errors: 0
 Number of warnings: 0

Writing NGD file "can_bd.ngd" ...

Writing NGDBUILD log file "can_bd.bld"...

APPENDIX 6

Map Report

Map report

Release 4.2i - Map E.35
Xilinx Mapping Report File for Design 'can_bd'

Design Information

Command Line : map -p xc2v1000-fg256-4 -cm area -k 4 -c 100 -tx off can_bd.ngd
Target Device : x2v1000
Target Package : fg256
Target Speed : -4
Mapper Version : virtex2 -- $Revision: 1.58 $
Mapped Date : Wed Apr 28 21:36:19 2004

Design Summary

```
  Number of errors:      0
  Number of warnings:    0
  Number of Slices:                 12 out of   5,120    1%
  Number of Slices containing
     unrelated logic:                0 out of      12    0%
  Number of Slice Flip Flops:       19 out of  10,240    1%
  Number of 4 input LUTs:            4 out of  10,240    1%
  Number of bonded IOBs:             7 out of     172    4%
  Number of GCLKs:                   1 out of      16    6%
  Number of DCMs:                    1 out of       8   12%
Total equivalent gate count for design: 7,179
Additional JTAG gate count for IOBs: 336
```

Table of Contents

Section 1 - Errors

Section 2 - Warnings

Section 3 - Informational

INFO:MapLib:354 - Virtex BUFG symbol "u11_u_bufg" (output signal=net5381) is

being retargetted to Virtex2 BUFGMUX with input tied to I0 and Select pin

tied to constant 0.

INFO:MapLib:62 - All of the external outputs in this design are using slew rate

limited output drivers. The delay on speed critical outputs can be

dramatically reduced by designating them as fast outputs in the schematic.

Section 4 - Removed Logic Summary

----

2 block(s) optimized away

Section 5 - Removed Logic

Optimized Block(s):

TYPE	BLOCK
GND	GND_I
VCC	VCC_I

To enable printing of redundant blocks removed and signals merged, set the

detailed map report option and rerun map.

Section 6 - IOB Properties

IOB Name	Type	Direction	IO Standard	Drive Strength	Slew Rate	Reg (s)	Resistor	IOB Delay
c_bus	IOB	OUTPUT	LVTTL	12	SLOW			
c_clk_output	IOB	OUTPUT	LVTTL	12	SLOW			
c_clock	IOB	INPUT	LVTTL					
c_dout	IOB	OUTPUT	LVTTL	12	SLOW			
c_reset	IOB	INPUT	LVTTL					
lock	IOB	OUTPUT	LVTTL	12	SLOW			
reset	IOB	OUTPUT	LVTTL	12	SLOW			

Section 7 - RPMs

Section 8 - Guide Report

Guide not run on this design.

Section 9 - Area Group Summary

No area groups were found in this design.

Section 10 - Modular Design Summary

Modular Design not used for this design.

APPENDIX 7

Place & Route Report

Place & Route report

Release 4.2i - Par E.35

Wed Apr 28 21:36:26 2004

par -f _par.rsp

Constraints file: can_bd.pcf

Loading design for application par from file par_temp.ncd.
"can_bd" is an NCD, version 2.37, device xc2v1000, package fg256, speed -4
Loading device for application par from file '2v1000.nph' in environment

C:/Xilinx.
Device speed data version: PRODUCTION 1.96 2002-01-02.

Resolved that IOB <c_dout> must be placed at site A8.
Resolved that IOB <c_clock> must be placed at site P9.
Resolved that IOB <c_bus> must be placed at site A7.
Resolved that IOB <c_reset> must be placed at site M4.
Resolved that IOB <reset> must be placed at site C5.
Resolved that IOB <c_clk_output> must be placed at site B8.
Resolved that IOB <lock> must be placed at site D5.

Device utilization summary:

Number of External IOBs	7 out of 172	4%
Number of LOCed External IOBs	7 out of 7	100%
Number of SLICEs	12 out of 5120	1%
Number of BUFGMUXs	1 out of 16	6%
Number of DCMs	1 out of 8	12%

Overall effort level (-ol): 2 (set by user)
Placer effort level (-pl): 2 (set by user)
Placer cost table entry (-t): 1
Router effort level (-rl): 2 (set by user)

```
Extra effort level (-xe):     0 (set by user)

Starting Clock Logic Placement.  REAL time: 7 secs

Placer score = 21
Finished Clock Logic Placement.  REAL time: 7 secs

Automatic resolution of clock placement was successful.
It was not necessary to constrain the placement of any of the logic driven by the global
clocks with the current clock placement.

###############################################################
## Automatic clock placement completed.
###############################################################

Starting clustering phase.  REAL time: 7 secs
Finished clustering phase.  REAL time: 7 secs

Starting Directed Placer.  REAL time: 8 secs
Placement pass 1 .

Placer score = 5610
Placer score = 5610
Finished Directed Placer.  REAL time: 8 secs

Starting Optimizing Placer.  REAL time: 8 secs
Optimizing
Swapped 9 comps.
Xilinx Placer [1]   5310   REAL time: 8 secs
Finished Optimizing Placer.  REAL time: 8 secs

Dumping design to file can_bd.ncd.

Total REAL time to Placer completion: 8 secs
Total CPU time to Placer completion: 5 secs

0 connection(s) routed; 70 unrouted active, 7 unrouted PWR/GND.
Starting router resource preassignment
Completed router resource preassignment. REAL time: 10 secs
Starting iterative routing.
Routing active signals.
........
End of iteration 1
77 successful; 0 unrouted; (0) REAL time: 12 secs
Constraints are met.
Total REAL time: 12 secs
```

Total CPU time: 8 secs
End of route. 77 routed (100.00%); 0 unrouted.
No errors found.
WARNING:Route:49 - The signal "GLOBAL_LOGIC0" has no loads so was not routed.

This design was run without timing constraints. It is likely that much better circuit performance can be obtained by trying either or both of the following:

- Enabling the Delay Based Cleanup router pass, if not already enabled
- Supplying timing constraints in the input design

Total REAL time to Router completion: 12 secs
Total CPU time to Router completion: 8 secs

Generating PAR statistics.

The Delay Summary Report

The Score for this design is: 5222

The Number of signals not completely routed for this design is: 0

The Average Connection Delay for this design is: 1.765 ns
The Maximum Pin Delay is: 4.448 ns
The Average Connection Delay on the 10 Worst Nets is: 2.291 ns

Listing Pin Delays by value: (ns)

d < 1.00	< d < 2.00	< d < 3.00	< d < 4.00	< d < 5.00	d >= 5.00
34	18	10	9	6	0

Dumping design to file can_bd.ncd.

All signals are completely routed.

Total REAL time to PAR completion: 13 secs
Total CPU time to PAR completion: 9 secs

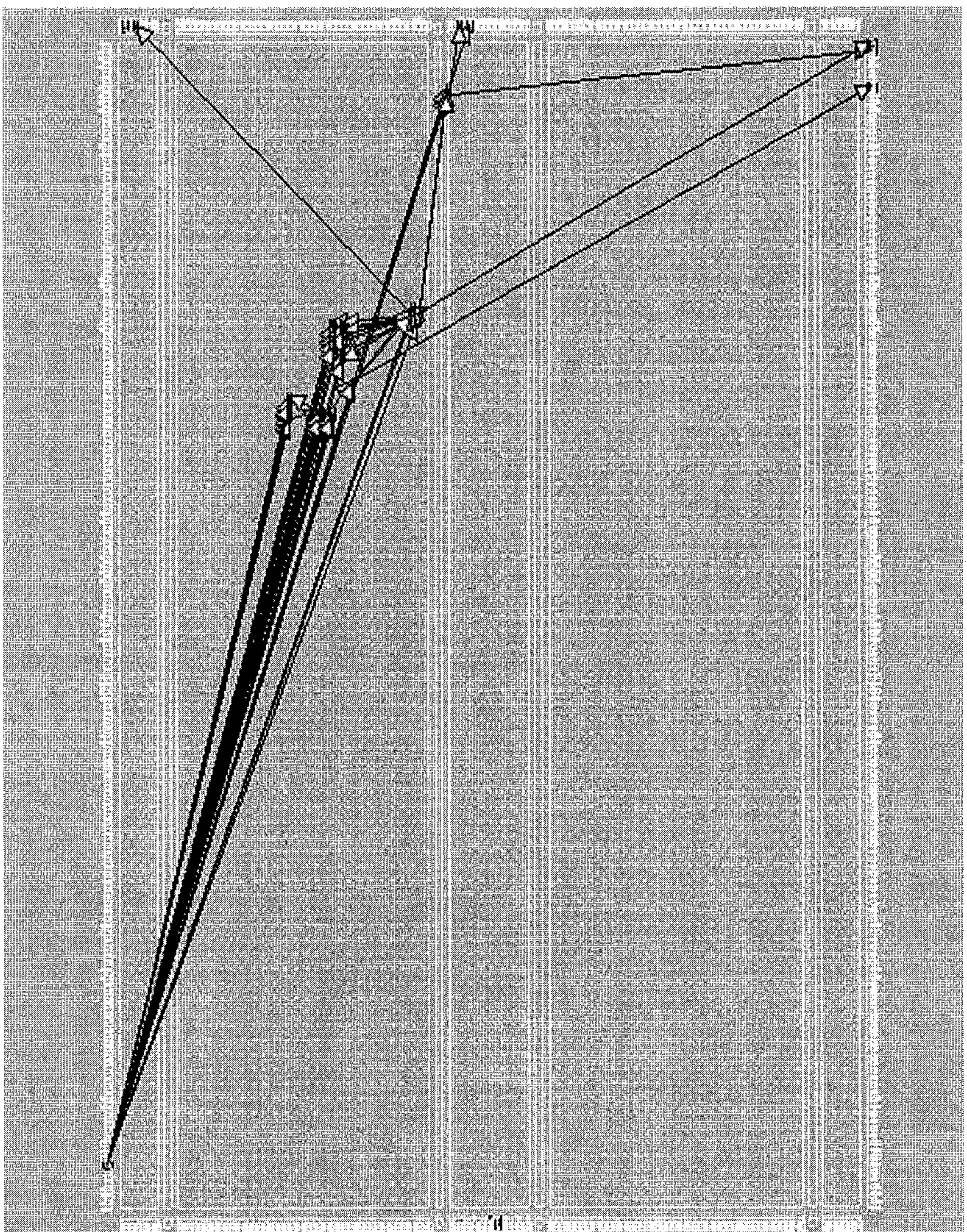
Placement: Completed - No errors found.
Routing: Completed - No errors found.

PAR done.

APPENDIX 8

FPGA Floorplan

FPGA Floorplan



APPENDIX 9

Pad Report

Pad report

Release 4.2i - Par E.35

Wed Apr 28 21:36:39 2004

Xilinx PAD Specification File
****

Input file: par_temp.ncd
Output file: can_bd.ncd
Part type: xc2v1000
Speed grade: -4
Package: fg256

Pinout by Signal Name:

Signal Name	Pin Name	Pin Number	Direction	IO Standard	IO Bank #	Drive (mA)	Slew Rate	Pullup Pulldown	IOB Delay	Voltage	Constraint
c_bus		A7	OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED
c_clk_output	GCLK5P	B8	OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED
c_clock	GCLK2P	P9	INPUT	LVTTL	4	12*	SLOW*	NONE**	NONE		LOCATED
c_dout	GCLK4S	A8	OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED
c_reset		M4	INPUT	LVTTL	6	12*	SLOW*	NONE**	NONE		LOCATED
lock		D5	OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED
reset		C5	OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED

Pinout by Pin Number:

Pin Number	Signal Name	Pin Name	Direction	IO Standard	IO Bank #	Drive (mA)	Slew Rate	Pullup Pulldown	IOB Delay	Voltage	Constraint
A1		GND		LVTTL*		12*	SLOW*	NONE**	***		
A2		PROG_B		LVTTL*		12*	SLOW*	NONE**	***		
A3		RSVD		LVTTL*		12*	SLOW*	NONE**	***		
A4		RSVD		LVTTL*		12*	SLOW*	NONE**	***		
A5			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
A6			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
A7	c_bus		OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED
A8	c_dout	GCLK4S	OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED
A9		GCLK3P		LVTTL*	1	12*	SLOW*	NONE**	***		
A10			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
A11			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
A12			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
A13		RSVD		LVTTL*		12*	SLOW*	NONE**	***		
A14		VBATT		LVTTL*		12*	SLOW*	NONE**	***		
A15		TCK		LVTTL*		12*	SLOW*	NONE**	***		
A16		GND		LVTTL*		12*	SLOW*	NONE**	***		
B1		VCCAUX		LVTTL*		12*	SLOW*	NONE**	***		
B2		GND		LVTTL*		12*	SLOW*	NONE**	***		
B3		HSWAP_EN		LVTTL*		12*	SLOW*	NONE**	***		
B4			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
B5			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
B6			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		

B7		VREF		LVTTL*	0	12*	SLOW*	NONE**	***		
B8	c_clk_output	GCLK5P	OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED
B9		GCLK2S		LVTTL*	1	12*	SLOW*	NONE**	***		
B10		VREF		LVTTL*	1	12*	SLOW*	NONE**	***		
B11			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
B12			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
B13			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
B14		TMS		LVTTL*		12*	SLOW*	NONE**	***		
B15		GND		LVTTL*		12*	SLOW*	NONE**	***		
B16		VCCAUX		LVTTL*		12*	SLOW*	NONE**	***		
C1			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
C2		TDI		LVTTL*		12*	SLOW*	NONE**	***		
C3		GND		LVTTL*		12*	SLOW*	NONE**	***		
C4			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
C5	reset		OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED
C6			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
C7			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
C8		GCLK6S		LVTTL*	0	12*	SLOW*	NONE**	***		
C9		GCLK1P		LVTTL*	1	12*	SLOW*	NONE**	***		
C10			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
C11			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
C12			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
C13			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
C14		GND		LVTTL*		12*	SLOW*	NONE**	***		
C15		TDO		LVTTL*		12*	SLOW*	NONE**	***		
C16			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***		
D1			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
D2			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
D3			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
D4		VCCINT		LVTTL*		12*	SLOW*	NONE**	***		
D5	lock		OUTPUT	LVTTL	0	12	SLOW	NONE**	***		LOCATED
D6		VREF		LVTTL*	0	12*	SLOW*	NONE**	***		
D7			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
D8		GCLK7P		LVTTL*	0	12*	SLOW*	NONE**	***		
D9		GCLK0S		LVTTL*	1	12*	SLOW*	NONE**	***		
D10			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
D11		VREF		LVTTL*	1	12*	SLOW*	NONE**	***		
D12			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
D13		VCCINT		LVTTL*		12*	SLOW*	NONE**	***		
D14			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***		
D15			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***		
D16			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***		
E1			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
E2			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
E3		VREF		LVTTL*	7	12*	SLOW*	NONE**	***		
E4			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
E5		VCCINT		LVTTL*		12*	SLOW*	NONE**	***		
E6			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
E7			UNUSED	LVTTL*	0	12*	SLOW*	NONE**	***		
E8		VCCO_0		LVTTL*		12*	SLOW*	NONE**	***	3.30	
E9		VCCO_1		LVTTL*		12*	SLOW*	NONE**	***	na	
E10			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
E11			UNUSED	LVTTL*	1	12*	SLOW*	NONE**	***		
E12		VCCINT		LVTTL*		12*	SLOW*	NONE**	***		
E13			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***		
E14		VREF		LVTTL*	2	12*	SLOW*	NONE**	***		
E15			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***		
E16			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***		
F1			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
F2			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
F3			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
F4			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
F5			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***		
F6		GND		LVTTL*		12*	SLOW*	NONE**	***		
F7		VCCO_0		LVTTL*		12*	SLOW*	NONE**	***	3.30	
F8		VCCO_0		LVTTL*		12*	SLOW*	NONE**	***	3.30	
F9		VCCO_1		LVTTL*		12*	SLOW*	NONE**	***	na	
F10		VCCO_1		LVTTL*		12*	SLOW*	NONE**	***	na	
F11		GND		LVTTL*		12*	SLOW*	NONE**	***		

F12			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
F13			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
F14			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
F15			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
F16			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
G1		VREF		LVTTL*	7	12*	SLOW*	NONE**	***			
G2			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***			
G3			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***			
G4			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***			
G5		VREF		LVTTL*	7	12*	SLOW*	NONE**	***			
G6		VCCO_7		LVTTL*		12*	SLOW*	NONE**	***	na		
G7		GND		LVTTL*		12*	SLOW*	NONE**	***			
G8		GND		LVTTL*		12*	SLOW*	NONE**	***			
G9		GND		LVTTL*		12*	SLOW*	NONE**	***			
G10		GND		LVTTL*		12*	SLOW*	NONE**	***			
G11		VCCO_2		LVTTL*		12*	SLOW*	NONE**	***	na		
G12		VREF		LVTTL*	2	12*	SLOW*	NONE**	***			
G13			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
G14			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
G15			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
G16		VREF		LVTTL*	2	12*	SLOW*	NONE**	***			
H1			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***			
H2			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***			
H3			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***			
H4			UNUSED	LVTTL*	7	12*	SLOW*	NONE**	***			
H5		VCCO_7		LVTTL*		12*	SLOW*	NONE**	***	na		
H6		VCCO_7		LVTTL*		12*	SLOW*	NONE**	***	na		
H7		GND		LVTTL*		12*	SLOW*	NONE**	***			
H8		GND		LVTTL*		12*	SLOW*	NONE**	***			
H9		GND		LVTTL*		12*	SLOW*	NONE**	***			
H10		GND		LVTTL*		12*	SLOW*	NONE**	***			
H11		VCCO_2		LVTTL*		12*	SLOW*	NONE**	***	na		
H12		VCCO_2		LVTTL*		12*	SLOW*	NONE**	***	na		
H13			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
H14			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
H15			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
H16			UNUSED	LVTTL*	2	12*	SLOW*	NONE**	***			
J1			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***			
J2			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***			
J3			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***			
J4			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***			
J5		VCCO_6		LVTTL*		12*	SLOW*	NONE**	***	na		
J6		VCCO_6		LVTTL*		12*	SLOW*	NONE**	***	na		
J7		GND		LVTTL*		12*	SLOW*	NONE**	***			
J8		GND		LVTTL*		12*	SLOW*	NONE**	***			
J9		GND		LVTTL*		12*	SLOW*	NONE**	***			
J10		GND		LVTTL*		12*	SLOW*	NONE**	***			
J11		VCCO_3		LVTTL*		12*	SLOW*	NONE**	***	na		
J12		VCCO_3		LVTTL*		12*	SLOW*	NONE**	***	na		
J13			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***			
J14			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***			
J15			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***			
J16			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***			
K1		VREF		LVTTL*	6	12*	SLOW*	NONE**	***			
K2			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***			
K3			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***			
K4			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***			
K5		VREF		LVTTL*	6	12*	SLOW*	NONE**	***			
K6		VCCO_6		LVTTL*		12*	SLOW*	NONE**	***	na		
K7		GND		LVTTL*		12*	SLOW*	NONE**	***			
K8		GND		LVTTL*		12*	SLOW*	NONE**	***			
K9		GND		LVTTL*		12*	SLOW*	NONE**	***			
K10		GND		LVTTL*		12*	SLOW*	NONE**	***			
K11		VCCO_3		LVTTL*		12*	SLOW*	NONE**	***	na		
K12		VREF		LVTTL*	3	12*	SLOW*	NONE**	***			
K13			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***			
K14			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***			
K15			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***			
K16		VREF		LVTTL*	3	12*	SLOW*	NONE**	***			
L1			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***			

L2			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
L3			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
L4			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
L5			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
L6		GND		LVTTL*		12*	SLOW*	NONE**	***		
L7		VCCO_5		LVTTL*		12*	SLOW*	NONE**	***	na	
L8		VCCO_5		LVTTL*		12*	SLOW*	NONE**	***	na	
L9		VCCO_4		LVTTL*		12*	SLOW*	NONE**	***	na	
L10		VCCO_4		LVTTL*		12*	SLOW*	NONE**	***	na	
L11		GND		LVTTL*		12*	SLOW*	NONE**	***		
L12			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
L13			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
L14			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
L15			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
L16			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
M1			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
M2			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
M3		VREF		LVTTL*	6	12*	SLOW*	NONE**	***		
M4	c_reset		INPUT	LVTTL	6	12*	SLOW*	NONE**	NONE		LOCATED
M5		VCCINT		LVTTL*		12*	SLOW*	NONE**	***		
M6			UNUSED	LVTTL*	5	12*	SLOW*	NONE**	***		
M7			UNUSED	LVTTL*	5	12*	SLOW*	NONE**	***		
M8		VCCO_5		LVTTL*		12*	SLOW*	NONE**	***	na	
M9		VCCO_4		LVTTL*		12*	SLOW*	NONE**	***	na	
M10			UNUSED	LVTTL*	4	12*	SLOW*	NONE**	***		
M11			UNUSED	LVTTL*	4	12*	SLOW*	NONE**	***		
M12		VCCINT		LVTTL*		12*	SLOW*	NONE**	***		
M13			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
M14		VREF		LVTTL*	3	12*	SLOW*	NONE**	***		
M15			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
M16			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
N1			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
N2			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
N3			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
N4		VCCINT		LVTTL*		12*	SLOW*	NONE**	***		
N5		D5/ALT_VRN_5		LVTTL*	5	12*	SLOW*	NONE**	***		
N6			UNUSED	LVTTL*	5	12*	SLOW*	NONE**	***		
N7			UNUSED	LVTTL*	5	12*	SLOW*	NONE**	***		
N8		GCLK4P		LVTTL*	5	12*	SLOW*	NONE**	***		
N9		GCLK3S		LVTTL*	4	12*	SLOW*	NONE**	***		
N10			UNUSED	LVTTL*	4	12*	SLOW*	NONE**	***		
N11			UNUSED	LVTTL*	4	12*	SLOW*	NONE**	***		
N12		D2/ALT_VRP_4		LVTTL*	4	12*	SLOW*	NONE**	***		
N13		VCCINT		LVTTL*		12*	SLOW*	NONE**	***		
N14			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
N15			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
N16			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
P1			UNUSED	LVTTL*	6	12*	SLOW*	NONE**	***		
P2		M1		LVTTL*		12*	SLOW*	NONE**	***		
P3		GND		LVTTL*		12*	SLOW*	NONE**	***		
P4		D7		LVTTL*	5	12*	SLOW*	NONE**	***		
P5		D4/ALT_VRP_5		LVTTL*	5	12*	SLOW*	NONE**	***		
P6			UNUSED	LVTTL*	5	12*	SLOW*	NONE**	***		
P7			UNUSED	LVTTL*	5	12*	SLOW*	NONE**	***		
P8		GCLK5S		LVTTL*	5	12*	SLOW*	NONE**	***		
P9	c_clock	GCLK2P	INPUT	LVTTL	4	12*	SLOW*	NONE**	NONE		LOCATED
P10			UNUSED	LVTTL*	4	12*	SLOW*	NONE**	***		
P11			UNUSED	LVTTL*	4	12*	SLOW*	NONE**	***		
P12		D3/ALT_VRN_4		LVTTL*	4	12*	SLOW*	NONE**	***		
P13		D0		LVTTL*	4	12*	SLOW*	NONE**	***		
P14		GND		LVTTL*		12*	SLOW*	NONE**	***		
P15		CCLK		LVTTL*		12*	SLOW*	NONE**	***		
P16			UNUSED	LVTTL*	3	12*	SLOW*	NONE**	***		
R1		VCCAUX		LVTTL*		12*	SLOW*	NONE**	***		
R2		GND		LVTTL*		12*	SLOW*	NONE**	***		
R3		M2		LVTTL*		12*	SLOW*	NONE**	***		
R4		D6		LVTTL*	5	12*	SLOW*	NONE**	***		
R5		VREF		LVTTL*	5	12*	SLOW*	NONE**	***		
R6		VREF		LVTTL*	5	12*	SLOW*	NONE**	***		

R7		VREF		LVTTL*	5	12*	SLOW*	NONE**	***			
R8		GCLK6P		LVTTL*	5	12*	SLOW*	NONE**	***			
R9		GCLK1S		LVTTL*	4	12*	SLOW*	NONE**	***			
R10		VREF		LVTTL*	4	12*	SLOW*	NONE**	***			
R11		VREF		LVTTL*	4	12*	SLOW*	NONE**	***			
R12		VREF		LVTTL*	4	12*	SLOW*	NONE**	***			
R13		D1		LVTTL*	4	12*	SLOW*	NONE**	***			
R14		DONE		LVTTL*		12*	SLOW*	NONE**	***			
R15		GND		LVTTL*		12*	SLOW*	NONE**	***			
R16		VCCAUX		LVTTL*		12*	SLOW*	NONE**	***			
T1		GND		LVTTL*		12*	SLOW*	NONE**	***			
T2		M0		LVTTL*		12*	SLOW*	NONE**	***			
T3		CS_B		LVTTL*	5	12*	SLOW*	NONE**	***			
T4		RDWR_B		LVTTL*	5	12*	SLOW*	NONE**	***			
T5			UNUSED	LVTTL*	5	12*	SLOW*	NONE**	***			
T6			UNUSED	LVTTL*	5	12*	SLOW*	NONE**	***			
T7			UNUSED	LVTTL*	5	12*	SLOW*	NONE**	***			
T8		GCLK7S		LVTTL*	5	12*	SLOW*	NONE**	***			
T9		GCLK0P		LVTTL*	4	12*	SLOW*	NONE**	***			
T10			UNUSED	LVTTL*	4	12*	SLOW*	NONE**	***			
T11			UNUSED	LVTTL*	4	12*	SLOW*	NONE**	***			
T12			UNUSED	LVTTL*	4	12*	SLOW*	NONE**	***			
T13		INIT_B		LVTTL*	4	12*	SLOW*	NONE**	***			
T14		DOUT		LVTTL*	4	12*	SLOW*	NONE**	***			
T15		PWRDWN_B		LVTTL*		12*	SLOW*	NONE**	***			
T16		GND		LVTTL*		12*	SLOW*	NONE**	***			

* Default value.
** This default Pullup/Pulldown value can be overridden in Bitgen.
*** The default IOB Delay is determined by how the IOB is used.

```
#
# To preserve the pinout above for future design iterations,
# simply invoke PIN2UCF from the command line or issue this command in the GUI.
# For Foundation ISE/Project Navigator - Run the process "Implement Design" ->
"Place-and-Route" -> "Back-annotate Pin Locations"
# For Design Manager - In the Design menu select "Lock Pins...
# The location constraints above will be written into your specified UCF file. (The
constraints
# listed below are in PCF format and cannot be directly used in the UCF file).
#
COMP "c_bus" LOCATE = SITE "A7" ;
COMP "c_clk_output" LOCATE = SITE "B8" ;
COMP "c_clock" LOCATE = SITE "P9" ;
COMP "c_dout" LOCATE = SITE "A8" ;
COMP "c_reset" LOCATE = SITE "M4" ;
COMP "lock" LOCATE = SITE "D5" ;
COMP "reset" LOCATE = SITE "C5" ;
#
```

APPENDIX 10

Asynchronous Delay Report

Asynchronous Delay report

Release 4.2i - Par E.35

Wed Apr 28 21:36:38 2004

File: can_bd.dly

The 20 Worst Net Delays are:

Max Delay (ns)	Netname
4.448	lock_OBUF
4.329	reset_OBUF
2.589	net5381
2.464	u2_do
2.171	c_bus_OBUF
1.598	u7_tmp<1>
1.586	u2_tmp<1>
1.582	u2_tmp<3>
1.291	u2_tmp<5>
1.257	u2_tmp<7>
1.194	N58
0.990	u7_tmp<7>
0.950	u2_tmp<0>
0.949	u2_tmp<4>
0.938	u7_tmp<0>
0.935	u7_tmp<4>
0.934	u7_tmp<2>
0.934	u2_tmp<6>
0.934	u7_tmp<6>
0.933	u2_tmp<2>

Net Delays

GLOBAL_LOGIC1
  PWR_VCC_0.VCCOUT
    0.172 u11_u_bufg.S

GLOBAL_LOGIC1_0
  PWR_VCC_1.VCCOUT
    0.070 u7_tmp<1>.BY

GLOBAL_LOGIC1_1
  PWR_VCC_2.VCCOUT
    0.070 u2_tmp<1>.BY

GLOBAL_LOGIC1_2

```
  PWR_VCC_3.VCCOUT
      0.115  u11_u_dcm.DSSEN
      0.151  u11_u_dcm.PSCLK
      0.115  u11_u_dcm.PSEN
      0.115  u11_u_dcm.PSINCDEC

N56
  N56.X
      0.408  u2_tmp<1>.CE
      0.408  u2_tmp<3>.CE
      0.408  u2_tmp<5>.CE
      0.389  u2_tmp<7>.CE

N58
  u1_tstate.Y
      1.194  u2_do.SR

c_bus_OBUF
  N56.Y
      2.171  c_bus.O1

c_clock_IBUFG
  c_clock.I
      0.798  u11_u_dcm.CLKIN

lock_OBUF
  u11_u_dcm.LOCKED
      4.448  lock.O1

net5381
  u11_u_bufg.O
      2.589  c_clk_output.O1
      1.366  u11_u_dcm.CLKFB
      1.097  u7_tmp<1>.CLK
      1.097  u7_tmp<3>.CLK
      1.096  u7_tmp<5>.CLK
      1.094  u7_tmp<7>.CLK
      1.093  u2_do.CLK
      1.093  u1_tstate.CLK
      1.098  u7_do.CLK
      1.101  u2_tmp<1>.CLK
      1.101  u2_tmp<3>.CLK
      1.101  u2_tmp<5>.CLK
      1.098  u2_tmp<7>.CLK

reset_OBUF
  c_reset.I
      3.675  reset.O1
      2.307  u11_u_dcm.RST
      3.432  u7_tmp<1>.CE
      3.723  u7_tmp<1>.SR
      3.432  u7_tmp<3>.CE
      3.723  u7_tmp<3>.SR
      2.991  u7_tmp<5>.CE
      3.065  u7_tmp<5>.SR
      4.034  u7_tmp<7>.CE
```

```
      3.730  u7_tmp<7>.SR
      3.721  u2_do.CE
      4.329  u1_tstate.SR
      3.721  u1_tstate.F4
      4.024  u1_tstate.G3
      4.034  u7_do.CE
      2.774  u7_do.SR
      2.732  u2_tmp<1>.SR
      2.732  u2_tmp<3>.SR
      2.732  u2_tmp<5>.SR
      4.229  N56.F2
      2.774  u2_tmp<7>.SR

u11_clk0_w
  u11_u_dcm.CLK0
      0.852  u11_u_bufg.I0

u1_I_tnext/O
  u1_tstate.X
      0.001  u1_tstate.DX

u1_tstate
  u1_tstate.XQ
      0.532  u1_tstate.F1
      0.569  u1_tstate.G1
      0.285  N56.F4

u2_do
  u2_do.YQ
      2.464  c_dout.O1
      0.325  u1_tstate.G4
      0.533  N56.F1
      0.570  N56.G1

u2_tmp<0>
  u2_tmp<1>.YQ
      0.950  u2_tmp<1>.BX

u2_tmp<1>
  u2_tmp<1>.XQ
      1.586  u2_tmp<3>.BY

u2_tmp<2>
  u2_tmp<3>.YQ
      0.933  u2_tmp<3>.BX

u2_tmp<3>
  u2_tmp<3>.XQ
      1.582  u2_tmp<5>.BY

u2_tmp<4>
  u2_tmp<5>.YQ
      0.949  u2_tmp<5>.BX

u2_tmp<5>
  u2_tmp<5>.XQ
```

```
      1.291  u2_tmp<7>.BY

u2_tmp<6>
  u2_tmp<7>.YQ
      0.934  u2_tmp<7>.BX

u2_tmp<7>
  u2_tmp<7>.XQ
      1.257  u2_do.BY

u7_do
  u7_do.YQ
      0.786  u1_tstate.G2
      0.641  N56.F3
      0.610  N56.G3

u7_tmp<0>
  u7_tmp<1>.YQ
      0.938  u7_tmp<1>.BX

u7_tmp<1>
  u7_tmp<1>.XQ
      1.598  u7_tmp<3>.BY

u7_tmp<2>
  u7_tmp<3>.YQ
      0.934  u7_tmp<3>.BX

u7_tmp<3>
  u7_tmp<3>.XQ
      0.693  u7_tmp<5>.BY

u7_tmp<4>
  u7_tmp<5>.YQ
      0.935  u7_tmp<5>.BX

u7_tmp<5>
  u7_tmp<5>.XQ
      0.693  u7_tmp<7>.BY

u7_tmp<6>
  u7_tmp<7>.YQ
      0.934  u7_tmp<7>.BX

u7_tmp<7>
  u7_tmp<7>.XQ
      0.990  u7_do.BY
```

APPENDIX 11

Post-Place & Route Static Timing Report

Post-Place & Route Static Timing Report

Release 4.2i - Trace E.35

trce -e 3 -l 3 -xml can_bd can_bd.ncd -o can_bd.twr can_bd.pcf

Design file: can_bd.ncd
Physical constraint file: can_bd.pcf
Device,speed: xc2v1000,-4 (PRODUCTION 1.96 2002-01-02)
Report level: error report

WARNING:Timing:2491 - No timing constraints found, doing default enumeration.

Timing constraint: Default period analysis

89 items analyzed, 0 timing errors detected.
Minimum period is   6.762ns.
Maximum delay is 10.042ns.

Timing constraint: Default net enumeration

32 items analyzed, 0 timing errors detected.
Maximum net delay is   4.448ns.

All constraints were met.

Data Sheet report:

All values displayed in nanoseconds (ns)

Setup/Hold to clock c_clock

Source Pad	Setup to clk (edge)	Hold to clk (edge)
c_reset	7.267(R)	0.000(R)

Clock c_clock to Pad

Destination Pad	clk (edge) to PAD
c_bus	8.830(R)
c_clk_output	6.533(X)
c_dout	8.069(R)

Clock to Setup on destination clock c_clock

Source Clock	Src:Rise Dest:Rise	Src:Fall Dest:Rise	Src:Rise Dest:Fall	Src:Fall Dest:Fall
c_clock	3.272			

Pad to Pad

Source Pad	Destination Pad	Delay
c_reset	reset	10.042

Timing summary:

Timing errors: 0 Score: 0

Constraints cover 89 paths, 32 nets, and 70 connections (100.0% coverage)

Design statistics:
  Minimum period: 6.762ns (Maximum frequency: 147.885MHz)
  Maximum combinational path delay: 10.042ns
  Maximum net delay: 4.448ns

Analysis completed Wed Apr 28 21:54:16 2004

APPENDIX 12

BitGen Report

BitGen report

Release 4.2i - Bitgen E.35

Loading design for application Bitgen from file can_bd.ncd.
"can_bd" is an NCD, version 2.37, device xc2v1000, package fg256, speed -4
Loading device for application Bitgen from file '2v1000.nph' in environment

C:/Xilinx.
Opened constraints file can_bd.pcf.

Wed Apr 28 22:02:14 2004

bitgen -w -g DebugBitstream:No -g CRC:Enable -g ConfigRate:4 -g CclkPin:PullUp -g M0Pin:PullUp -g M1Pin:PullUp -g M2Pin:PullUp -g ProgPin:PullUp -g DonePin:PullUp -g DriveDone:No -g PowerdownPin:PullUp -g TckPin:PullUp -g TdiPin:PullUp -g TdoPin:PullNone -g TmsPin:PullUp -g UnusedPin:PullUp -g UserID:0xFFFFFFFF -g DCMShutDown:Disable -g DisableBandgap:No -g StartUpClk:CClk -g DONE_cycle:4 -g GTS_cycle:5 -g GWE_cycle:6 -g LCK_cycle:NoWait -g Match_cycle:NoWait -g Security:None -g Persist:No -g DonePipe:No -g Encrypt:No can_bd.ncd

Summary of Bitgen Options:

Option Name	Current Setting
Compress	(Not Specified)*
Readback	(Not Specified)*
CRC	Enable**
DebugBitstream	No**
ConfigRate	4**
StartupClk	Cclk**
DCMShutdown	Disable**
DisableBandgap	No**
CclkPin	Pullup**

DonePin	Pullup**
HswapenPin	Pullup*
M0Pin	Pullup**
M1Pin	Pullup**
M2Pin	Pullup**
PowerdownPin	Pullup**
ProgPin	Pullup**
TckPin	Pullup**
TdiPin	Pullup**
TdoPin	Pullnone
TmsPin	Pullup**
UnusedPin	Pullup
GWE_cycle	6**
GTS_cycle	5**
LCK_cycle	NoWait**
Match_cycle	NoWait
DONE_cycle	4**
Persist	No**
DriveDone	No**
DonePipe	No**
Security	None**
UserID	0xFFFFFFFF**
Encrypt	No**

Key0	pick*
Key1	pick*
Key2	pick*
Key3	pick*
Key4	pick*
Key5	pick*
Keyseq0	M*
Keyseq1	M*
Keyseq2	M*
Keyseq3	M*
Keyseq4	M*
Keyseq5	M*
KeyFile	(Not Specified)*
StartKey	0*
StartCBC	pick*
Binary	No*

* Default setting.

** The specified setting matches the default setting.

Running DRC.
WARNING:DesignRules:366 - Netcheck: Sourceless and loadless. Net GLOBAL_LOGIC0 has no pin.
DRC detected 0 errors and 1 warnings.
Creating bit map...
Saving bit stream in "can_bd.bit".
Bitstream generation is complete.

APPENDIX 13

User Constraint File (UCF)

```
##################################################
#    BASIC UCF SYNTAX EXAMPLES V2.1.5     #
##################################################
#
#           TIMING SPECIFICATIONS
#
# Timing specifications can be applied to the entire device (global) or to
# specific groups of login in your PLD design (called "time groups').
# The time groups are declared in two basic ways.
#
# Method 1: Based on a net name, where 'my_net' is a net that touchs all the
#         logic to be grouped in to 'logic_grp'. Example:
#NET my_net TNM_NET = logic_grp ;
#
# Method 2: Group using the key word 'TIMEGRP' and declare using the names of
#         logic in your design. Example:
#TIMEGRP group_name = FFS ("U1/*");
#         creates a group called 'group_name' for all flip-flops with in
#         the hierarchical block called U1. Wildcards are valid.
#
# Grouping is very important because it lets you tell the software which parts
# of a design run at which speeds.  For the majority of the designs with only
# one clock the very simple global constraints.
#
# The type of grouping constraint you use can vary depending on the synthesis
# tools you are using.  For example, Synplicity does well with Method 1, while
# FPGA Express does beter with Method 2.
#
#
#################################################################
# Internal to the device clock speed specifications - Tsys #
#################################################################
#
# data       _________       /^^^^^\       _________   out
# ----------| D   Q |-----{ LOGIC } -----| D   Q |------
#        |     |    \vvvvv/      |     |
#      ---|> CLK |               ---|> CLK |
# clock |  ---------            | ---------
# ------------------------------------
#
# ---------------
# Single Clock
# ---------------
#
# ----------------
# PERIOD TIME-SPEC
# ----------------
# The PERIOD spec. covers all timing paths that start or end at a
# register, latch, or synchronous RAM which are clocked by the reference
# net (excluding pad destinations).  Also covered is the setup
# requirement of the synchronous element relative to other elements
# (ex. flip flops, pads, etc...).
# NOTE:  The default unit for time is nanoseconds.
#
#NET clock PERIOD = 50ns ;
#
```

```
#     -OR-
#
# ------------------
# FROM:TO TIME-SPECs
# ------------------
# FROM:TO style timespecs can be used to constrain paths between time
# groups. NOTE: Keywords: RAMS, FFS, PADS, and LATCHES are predefined
# time groups used to specify all elements of each type in a design.
#TIMEGRP RFFS = RISING FFS ("*"); // creates a rising group called RFFS
#TIMEGRP FFFS = FALLING FFS ("*"); // creates a falling group called FFFS
#TIMESPEC TSF2F  = FROM : FFS  : TO : FFS  : 50 ns; // Flip-flips with the same edge
#TIMESPEC TSR2F  = FROM : RFFS  : TO : FFFS  : 25 ns; // rising edge to falling edge
#TIMESPEC TSF2R  = FROM : FFFS  : TO : RFFS  : 25 ns; // falling edge to rising edge
#
# ---------------
# Multiple Clocks
# ---------------
# Requires a combination of the 'Period' and 'FROM:TO' type time specifications
#NET clock1 TNM_NET = clk1_grp ;
#NET clock2 TNM_NET = clk2_grp ;
#
#TIMESPEC TS_clk1 = PERIOD : clk1_grp : 50 ;
#TIMESPEC TS_clk2 = PERIOD : clk2_grp : 30 ;
#TIMESPEC TS_ck1_2_ck2 = FROM : clk1_grp : TO : clk2_grp : 50 ;
#TIMESPEC TS_ck2_2_ck1 = FROM : clk2_grp : TO : clk1_grp : 30 ;
#
#
##############################################################
# CLOCK TO OUT specifications - Tco                  #
##############################################################
#
# from       __________       /^^^^^\      --------\
# ----------| D   Q |-----{ LOGIC } -----| Pad    >
# PLD       |      |     \vvvvv/      --------/
#       ---|> CLK  |
# clock  |  ---------
# --------
#
# ----------------
# OFFSET TIME-SPEC
# ----------------
# To automatically include clock buffer/routing delay in your
# clock-to-out timing specifications, use OFFSET constraints .
# For an output where the maximum clock-to-out (Tco) is 25 ns:
#NET out_net_name OFFSET = OUT 25 AFTER clock_net_name ;
#
#     -OR-
#
# ------------------
# FROM:TO TIME-SPECs
# ------------------
#TIMESPEC TSF2P  = FROM : FFS  : TO : PADS  : 25 ns;
# Note that FROM: FFS : TO: PADS constraints start the delay analysis
# at the flip flop itself, and not the clock input pin. The recommended
# method to create a clock-to-out constraint is to use an OFFSET constraint.
#
```

```
#
#################################################################
# Pad to Flip-Flop speed specifications - Tsu           #
#################################################################
#
# ------\       /^^^^^\      _________   into PLD
# |pad  >-------{ LOGIC } -----| D   Q |------
# ------/       \vvvvv/       |        |
#                          ---|> CLK  |
# clock                       | ---------
# ----------------------------
#
# ----------------
# OFFSET TIME-SPEC
# ----------------
# To automatically account for clock delay in your input setup timing
# specifications, use OFFSET constraints.
# For an input where the maximum setup time is 25 ns:
#NET in_net_name OFFSET = IN 25 BEFORE clock_net_name ;
#
#     -OR-
#
# ------------------
# FROM:TO TIME-SPECs
# ------------------
#TIMESPEC TSP2F  = FROM : PADS  : TO : FFS   : 25 ns;
# Note that FROM: PADS : TO: FFS constraints do not take into account any
# delay for the clock path.  The recommended method to create an input
# setup time constraint is to use an OFFSET constraint.
#
#
#################################################################
# Pad to Pad speed specifications - Tpd              #
#################################################################
#
# ------\       /^^^^^\      -------\
# |pad  >-------{ LOGIC } -----| pad  >
# ------/       \vvvvv/      -------/
#
# ------------------
# FROM:TO TIME-SPECs
# ------------------
#TIMESPEC TSP2P  = FROM : PADS  : TO : PADS  : 125 ns;
#
#
#################################################################
# Other timing specifications                    #
#################################################################
#
# -------------
# TIMING IGNORE
# -------------
# If you can ignore timing of paths, use Timing Ignore (TIG). NOTE: The
# "*" character is a wild-card which can be used for bus names.  A "?"
# character can be used to wild-card one character.
# Ignore timing of net reset_n:
```

```
#NET : reset_n : TIG ;
#
# Ignore data_reg(7:0) net in instance mux_mem:
#NET : mux_mem/data_reg* : TIG ;
#
# Ignore data_reg(7:0) net in instance mux_mem as related to a TIMESPEC
# named TS01 only:
#NET : mux_mem/data_reg* : TIG = TS01 ;
#
# Ignore data1_sig and data2_sig nets:
#NET : data?_sig : TIG ;
#
# ---------------
# PATH EXCEPTIONS
# ---------------
# If your design has outputs that can be slower than others, you can
# create specific timespecs similar to this example for output nets
# named out_data(7:0) and irq_n:
#TIMEGRP slow_outs = PADS(out_data* : irq_n) ;
#TIMEGRP fast_outs = PADS : EXCEPT : slow_outs ;
#TIMESPEC TS08 = FROM : FFS : TO : fast_outs : 22 ;
#TIMESPEC TS09 = FROM : FFS : TO : slow_outs : 75 ;
#
# If you have multi-cycle FF to FF paths, you can create a time group
# using either the TIMEGRP or TNM statements.
#
# WARNING:  Many VHDL/verilog synthesizers do not predictably name flip
# flop Q output nets.  Most synthesizers do assign predictable instance
# names to flip flops, however.
#
# TIMEGRP example:
#TIMEGRP slowffs = FFS(inst_path/ff_q_output_net1* :
#inst_path/ff_q_output_net2*);
#
# TNM attached to instance example:
#INST inst_path/ff_instance_name1_reg* TNM = slowffs ;
#INST inst_path/ff_instance_name2_reg* TNM = slowffs ;
#
# If a FF clock-enable is used on all flip flops of a multi-cycle path,
# you can attach TNM to the clock enable net.  NOTE:  TNM attached to a
# net "forward traces" to any FF, LATCH, RAM, or PAD attached to the
# net.
#NET ff_clock_enable_net TNM = slowffs ;
#
# Example of using "slowffs" timegroup, in a FROM:TO timespec, with
# either of the three timegroup methods shown above:
#TIMESPEC TS10 = FROM : slowffs : TO : FFS : 100 ;
#
# Constrain the skew or delay associate with a net.
#NET any_net_name MAXSKEW = 7 ;
#NET any_net_name MAXDELAY = 20 ns;
#
#
# Constraint priority in your .ucf file is as follows:
#
#   highest 1.  Timing Ignore (TIG)
```

```
#            2. FROM : THRU : TO specs
#          3. FROM : TO specs
#   lowest  4. PERIOD specs
#
# See the on-line "Library Reference Guide" document for
# additional timespec features and more information.
#
#
######################################################################
#                                                                    #
#      LOCATION and ATTRIBUTE SPECIFICATIONS                         #
#                                                                    #
######################################################################
# Pin and CLB location locking constraints                  #
######################################################################
#
# -----------------------
# Assign an IO pin number
# -----------------------
#INST io_buf_instance_name  LOC = P110 ;
#NET io_net_name  LOC = P111 ;
#
# -----------------------
# Assign a signal to a range of I/O pins
# -----------------------
#NET "signal_name" LOC=P32, P33, P34;
#
# -----------------------
# Place a logic element(called a BEL) in a specific CLB location.  BEL = FF, LUT, RAM, etc...
# -----------------------
#INST instance_path/BEL_inst_name  LOC = CLB_R17C36 ;
#
# -----------------------
# Place CLB in rectangular area from CLB R1C1 to CLB R5C7
# -----------------------
#INST /U1/U2/reg<0> LOC=clb_r1c1:clb_r5c7;
#
# -----------------------
# Place Heirarchial logic block in rectangular area from CLB R1C1 to CLB R5C7
# -----------------------
#INST /U1* LOC=clb_r1c1:clb_r5c7;
#
# -----------------------
# Prohibit IO pin P26 or CLBR5C3 from being used:
# -----------------------
#CONFIG PROHIBIT = P26 ;
#CONFIG PROHIBIT = CLB_R5C3 ;
# Config Prohibit is very important for frocing the software to not use critical
# configuration pins like INIT or DOUT on the FPGA.  The Mode pins and JTAG
# Pins require a special pad so they will not be availabe to this constraint
#
# -----------------------
# Assign an OBUF to be FAST or SLOW:
# -----------------------
#INST obuf_instance_name FAST ;
#INST obuf_instance_name SLOW ;
```

```
#
# ------------------------
# FPGAs only:  IOB input Flip-flop delay specifcation
# ------------------------
# Declare an IOB input FF delay (default = MAXDELAY).
# NOTE:  MEDDELAY/NODELAY can be attached to a CLB FF that is pushed
# into an IOB by the "map -pr i" option.
#INST input_ff_instance_name MEDDELAY ;
#INST input_ff_instance_name NODELAY ;
#
# ------------------------
# Assign Global Clock Buffers Lower Left Right Side
# ------------------------
# INST gbufl LOC=SSW
#
# #
NET "c_clock" LOC = "P9";
NET "c_reset" LOC = "M4";
NET "c_dout" LOC = "C4";
NET "c_bus" LOC = "A7";
NET "c_clk_output" LOC = "D5";
NET "reset" LOC = "A8";
NET "bus_status" LOC = "D16";
NET "can_b_out" LOC = "E13";
NET "enable_shifter" LOC = "C16";
```

APPENDIX 14

- FPGA Board Layout
- Virtex II Xilinx XC2V100 Demo Board Caption

Layout of Xilinx XC2V100 FPGA Demo Board

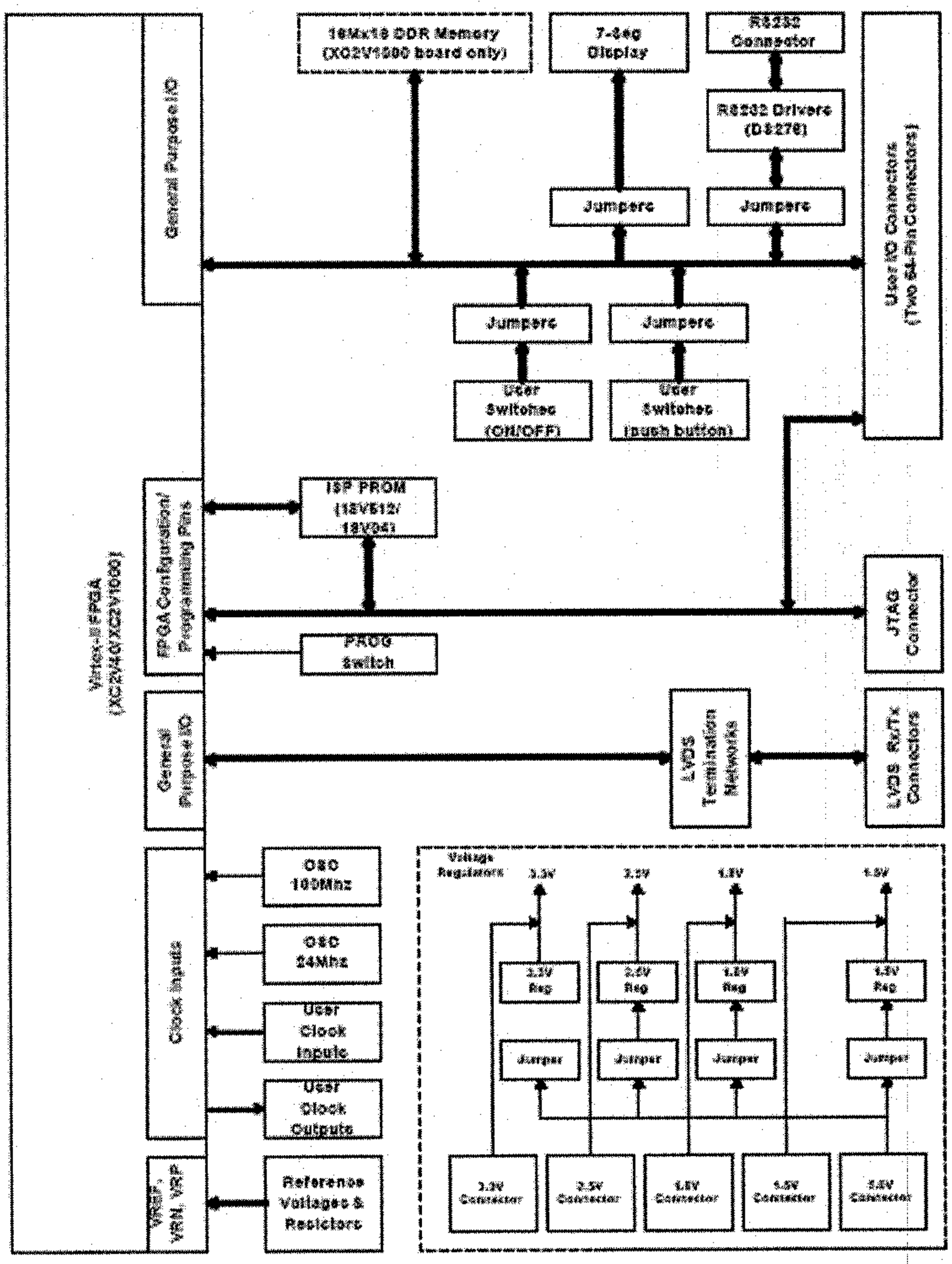


Figure 1 - XC2V40/XC2V1000 Reference Board Block Diagram

XILINX XC2V100 FPGA Demo Board